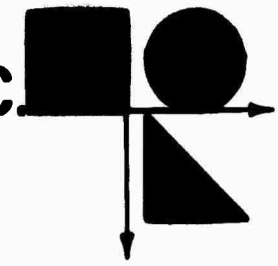
SECOND SEMI-ANNUAL TECHNICAL REPORT
(13 July 1970 - 12 January 1971)
FOR THE PROJECT
COMPILER DESIGN FOR THE ILLIAC IV

Massachusetts
COMPUTER ASSOCIATES
division of
APPLIED DATA RESEARCH, INC.



12

DISCLAIMER NOTICE

THIS DOCUMENT IS THE BEST QUALITY AVAILABLE.

COPY FURNISHED CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

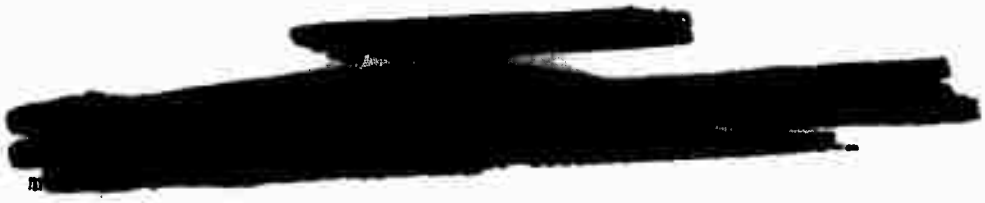
MASSACHUSETTS COMPUTER ASSOCIATES
A DIVISION OF APPLIED DATA RESEARCH, INC.
LAKESIDE OFFICE PARK • WAKEFIELD, MASSACHUSETTS 01880 • (617) 245-9540

SECOND SEMI-ANNUAL TECHNICAL REPORT
(13 July 1970 - 12 January 1971)
FOR THE PROJECT
COMPILER DESIGN FOR THE ILLIAC IV

Principal Investigator and Project Leader:
Robert E. Millstein Phone (617) 245-9540

ARPA Order Number ARPA 1554
Program Code Number 0D30
Contractor: Applied Data Research, Inc.
Contract No.: DAHC04 70 C 0023
Effective Date: 13 January 1970
Expiration Date: 12 October 1971
Amount: $303,012.50

Sponsored by
Advanced Research Projects Agency
ARPA Order No. 1554



ABSTRACT

The ILLIAC FORTRAN Compiler may be characterized as a series of transformations on the source input stream. FORTRAN code is transformed into a representation which exploits ILLIAC parallelism. This transformation is accomplished by detecting individual statements within DO loops which may be executed in parallel for values of the DO indices, and determining an ordering which preserves data dependencies. While the result of this effort affords ILLIAC parallelism, it is insensitive to two major characteristics of ILLIAC hardware: an enormous disk latency, and an ability to overlap execution of sequential and parallel components of the hardware. In order to fully exploit the capabilities of the ILLIAC, two more transformations are effected. First, code is restructured to minimize the effect of disk latency. Second, operations are allocated to maximize CU-PE overlap. At this stage it is appropriate to generate ILLIAC code.

TABLE OF CONTENTS

I. INTRODUCTION

The basic architecture of the ILLIAC IV Fortran Compiler may be characterized as a series of transformations (each representing a compilation phase) on the FORTRAN source input string. The first phase parses the input stream and generates a flow graph in which each statement is represented as a node.

The parsed string and flow graph depict an execution process which has been coded sequentially. It may contain operations which exhibit ILLIAC-exploitable parallelism. Detection of ILLIAC exploitable parallelism involves detecting individual statements within DO loops which may be executed in parallel for values of the DO indices, and determining a statement ordering which permits parallel execution without altering data dependencies. Chapter II, which examines parallelism detection is primarily concerned with data dependency and statement ordering. Appendix A provides a technique for detecting data dependencies in an arbitrarily complex flow graph.

By means of the parallelism algorithm, the flow graph is appropriately transformed. The parsed input stream is replaced by N-address macros which permit symbolic references (i.e., A(I,J+2) is a legal address).

At this point is is feasible to examine storage requirements with respect to I/O demands. Chapter III is concerned with optimizing I/O for large arrays. If this procedure dictates a new statement ordering, the appropriate transformations are effected. The examination of FORTRAN statement orderings provided an insight into the nature of partial orderings. These observations are contained in Appendix B.

After reordering in order to minimize I/O latency, array reference macros are expanded according to the array methodology described in the First Semi-Annual Report. The resulting pseudo code may be further optimized to take advantage of CU-PE execution overlap. Chapter IV defines ILLIAC optimization goals with respect to overlap, and provides a method for allocating instructions between CU and a single PE for sequential code. Appendix C specifies the upper boundary for execution savings in an overlap optimization effort.

At this state it is appropriate to generate ILLIAC IV code.

II. THE DETECTION OF PARALLELISM IN DO LOOPS

This section consists of two distinct parts. The first of these represents an overview of the nature and aims of the procedure for detecting parallelism in DO loops exploitable on the ILLIAC IV. The second is concerned with a particular aspect of this procedure: It is a discussion of the problem of data dependency detection in certain restricted but basic situations involving nested DO loops and multi-dimensioned arrays.

In the discussion of data dependency detection in both the present and previous semi-annual reports, relatively simple flow logic has been assumed for clarity of explication. In actual practice, of course, such simplicity is not likely to be the rule. Hence, included in Appendix A is the basis of a general algorithm, based largely on p-graph theory, for detecting data dependency in loops with any degree of control complexity.

PART 1: AN OVERVIEW

The goal of the parallelism procedure is the examination of FORTRAN DO loops in order to determine how much of the calculation, if any, can be done in parallel on the DO index variable. Manipulation of code will be performed whenever possible to maximize the amount of parallel computation. The analysis to accomplish the above will necessarily be restricted as to the complexity of code it can deal with; that is, in some cases (hopefully a small percentage) no attempt will be made to manipulate the code to effect parallelism.

RESTRICTIONS

1. The analysis of DO loops will extend to detection of parallelism on the index variable along 2 dimensions in a DO nest. In a DO loop nested 3 or more levels deep, code is analyzed for parallelism in the innermost DO variable, then in the next nesting DO loop variable, etc., until parallelism along 2 dimensions has been found. Along all

other dimensions, sequential execution is assumed. Example:

```
DO   I
A(I) =
DO   J
B(I,J) =
DO   K
C(I,K) =
DO   L
D(I,K,L) =
```

The L loop is the only one nested 3 or more deep. The code in this loop is analyzed first with respect to the L variable, then K, then I. If parallelism is found along the L and K dimensions, say, then with respect to the I dimension the code will be executed sequentially. Outside the L loop, code is examined for parallelism along 1 or 2 dimensions, as the case may be.

2. At present, internal (non-DO) cycles in a DO loop are considered as a special case: a cycle will be examined for parallelism (in the DO variable) as it stands; no attempt will be made to manipulate the code to make parallelism possible. In relation to the rest of the code in the loop, a cycle (or nest of cycles) will be treated as a "black box" containing only a list of definitions and references.

3. Subscripted variables. The procedure is based heavily on the analysis and ordering of the subscripts of array variables, primarily for the determination of inter-loop data dependencies. Only cases having subscripts of the standard linear form kI+c where I is the DO variable will be fully analyzed. Otherwise, it will be assumed that nothing is known about the relative value of a subscript and the "worst possible case" is assumed. This in general will force sequential execution of the affected code. A possible exception to the above restriction (since it appears to occur commonly) is a subscript which is a linear form in a non-DO variable which is, however, easily detected to be a linear form in the DO variable.

4. So far as the analysis described here is concerned, a case of "detected parallelism" is a piece of code for which it has been determined that the associated data dependencies are such that each statement can be executed simultaneously for all values of the index

variable. No representation is made, however, that the data for the computations can be presented simultaneously to the processing elements. That is, the analysis looks for concurrency of operator execution, but not operand fetching. In the statements:

$$A(I) = B(I) + C(I^2)$$
$$Z(I) = Y(I) + FUNCT(I)$$

it may be possible to perform the addition and store operations in parallel (on I). However, the fetching of elements of C in the first statement may require a complex sequence of instructions. In the second statement, calculation of the function FUNCT might require all processors; the values FUNCT(I) would then have to be generated sequentially.

PARALLELISM PROCEDURE

Outline:

1. Determination of data dependencies
2. Stating the ordering relations for parallel execution
   - Causal chains
   - Branches and merges
   - Cycles
3. Determination of optimal total ordering to minimize overwrite

1. Determination of Data Dependencies

Assume a technique equivalent to the p-graph algorithm to be applied to the loop code. For input to the algorithm, a "p-graph" for each variable (or uniquely subscripted variable) is represented; the nodes of the graph correspond generally to the uses and definitions of variables; additional nodes for merge points and the entry and exit points are supplied. Application of the algorithm gives all data dependencies for non-subscripted variables, that is, a variable use is explicitly related to one or more "circled" nodes which might have generated its present value. (It is assumed that the algorithm keeps track of all circled nodes associated with a merge node.)

For subscripted variables, the algorithm gives all intra-loop dependencies; the determination of inter-loop dependencies is more complex. If the graph for a subscripted variable shows a use of its initial value (either directly or via a merge), then it must be determined if this value could have been generated in a previous iteration of the loop. To do this, the graphs for the same array variable are assigned indices according to descending value of their literal subscripts*. For example, if array variable A appears in a loop on index I with subscripts I-1, I, I+1, then the A(I+1) graph is assigned 1, A(I) graph 2, etc. To find a possible generation in a previous loop iteration for a subscripted variable use with index n, the graphs corresponding to indices n-1, n-2, etc., are examined in turn. The first graph encountered having a non-initial value at its exit node gives an inter-loop dependency, i.e., the value was generated at the node(s) generating the exit value. If the node is a merge node with an initial value as input, the search for inter-loop dependencies is continued, halting finally when only non-initial values are encountered, or when all the p-graphs have been examined. It should be noted that this technique is made feasible by the observation that although the number of literal subscript expressions appearing in the source text within the range of the loop is in principle unlimited, it is in fact usually rather small.

Subscripting Non-Subscripted Variables

Non-subscripted variables will have to be subscripted in some cases in order to execute statements in parallel:

```
      DO  I
1     C = A(I)/D
2     B(I) = C * FUNCT(C)
```

In this example, C would have to be replaced by a vector, say C(I), in order to execute all definitions of C in parallel in statement 1. Both references to C in statement 2 would of course also be replaced by C(I). The variable D on the other hand need only be "broadcast" simultaneously to all processors since its value is the same for all I. In general, a non subscripted variable (and its dependencies) will be subscripted when it is defined in an assignment statement whose right

---

*Only subscripts of standard form kI+c will be ordered. Other subscripts are considered "indeterminate", i.e., possibly having any index value.

hand side is (directly or indirectly) a function of the DO variable. The subscripting will take two forms. For intra-loop dependencies, as in the above example, the subscript assigned is identical for the definition and all its references. For inter-loop dependencies, that is, when the value for a use of the variable was generated in the previous iteration--for example:

```
      DO  I
1     B(I) = C * FUNCT(C)
2     C = A(I+1)/D
```

the use will be assigned a subscript value 1 less than its definition. The above example may be reformulated:

```
      DO  I
1     B(I) = C(I-1) * FUNCT(C(I-1))
2     C(I) = A(I+1)/D
```

Assuming the first element of the vector C initialized to the value of C before entry into the loop, the above formulation is equivalent. Of course,for parallel execution of this loop, statements 1 and 2 must be reversed (see next section).

2. Ordering Relations for Parallel Execution

The condition for parallel execution is that in inter-loop dependencies, all value generations relevant to a variable use precede that use. To find out if it is possible to rearrange the code in the loop to meet this condition and at the same time preserve the essential data dependencies, a set of precedence relations might be constructed as follows. Assume that p-graph nodes have been numbered so that corresponding nodes on different graphs have the same number. Represent each data dependency by the relation: $n1 < n2$ where n2 is the node number of a variable use and n1 the node where its value was generated. (A relation for each possible generation, if more than one, must be stated.) If the total set of relations is examined and found consistent, i.e., if there are no cycles, then the DO loop can be made to be executed in parallel. There are easy techniques applicable to Boolean precedence matrices for detecting cycles and for determining total orderings from the given partial ordering [1,2,3].

If cycles are found in the ordering, then there are dependencies which represent real causal chains in the DO loop, for example:

```
A(I) = A(I-1)
```

which must be executed sequentially. The following examples illustrate another causal chain and a similar case which is not a chain:

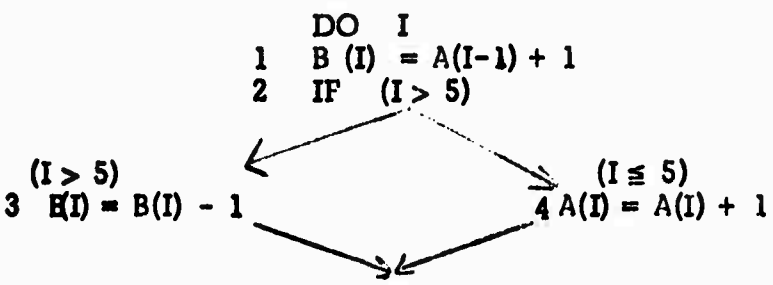
```
     DO I                        Ordering Relations
1    X (I) = A(I-1)              X:  1 < 2
2    A (I) = X(I)*2              A:  2 < 1

     DO I
1    X (I) = A(I-1)              A:  2 < 1
2    A(I)  = Y(I)                X, Y:  no dependencies
```

In general, only maximal cycles will be considered: the sequence of operations represented by each cycle will have to be executed sequentially (in the original order). However, all other operations in the DO loop (if any), can be performed in parallel. A total ordering can be determined by considering each cycle a single node and restating the ordering restrictions accordingly.

Branches and Merges

Rearrangement of code must necessarily take into account the flow logic of the loop. If the loop contains no cycles - only branches and merges - the problem is simple. Assume that in general all operations are associated with "mode sets", that is, with data words set to indicate the values of the DO variable over which an operation is to be defined. (If the loop contained only "straight line" code, all mode sets would conceptually be set to the entire DO index range.) Assume that at execution of an IF statement all relevant mode sets are set appropriately. Then it is only necessary to add to the data dependency ordering relations the conditions that setting of mode sets precede the operations dependent on them.

Example:



Assume statement 2 sets mode set 1 for I > 5 and mode set 2 for I ≦ 5. Statement 3 is associated with mode set 1, statement 4 with mode set 2. The ordering relations are:

A:	4 < 1
B:	1 < 3
Mode sets:	2 < 3
	2 < 4

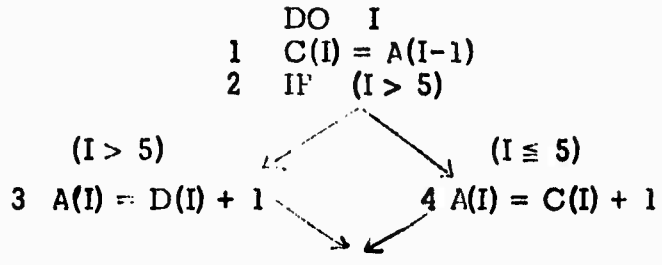
This gives the total ordering:

$$2 < 4 < 1 < 3$$

The loop is therefore executable in parallel as follows:

	Mode Sets
2 IF (I > 5)	All I
Sets mode sets 1 and 2	
4 A(I) = A(I) + 1	I in mode set 2
1 B(I) = A(I-1) + 1	All I
3 B(I) = B(I) - 1	I in mode set 1

Example:



Assume mode sets as before. The ordering relations are:

A:	4 < 1
	3 < 1
C:	1 < 4
Mode Sets:	2 < 3
	2 < 4

Total ordering:

2 < 3 < cycle (1,4)

The cycle forces sequential execution of the pair (1,4), but 2 and 3 can be executed in parallel:

	Mode Sets
2 IF (I > 5)	All I
Sets mode sets 1,2	
3 A(I) = D(I) + 1	I in mode set 1

Sequential loop:

1 C(I) = A(I-1)	All I
4 A(I) = C(I) + 1	I in mode set 2

Cycles

After data dependencies have been determined, internal (nests of) cycles are examined. If there is found to be any inter-loop dependency within a cycle, the loop cannot be executed in parallel (on the DO variable) as it stands:

```
      DO   10  I
      J =   1
1     A(I) = A(I) + 1          <-+
      B(I) = B(I) + A(I-1)       |
      J = J + 1                  |
      IF  (J < 1)  GO TO 1     --+
10    CONTINUE
```

In this example, A(I-1) is dependent on the final definition of A(I), that is, the value at completion of the internal cycle. In this case, the cycle could be split to permit parallel execution:

```
      DO   10  I
      J =   1
1     A(I) = A(I) + 1          <-+
      J = J + 1                  |
      IF  (J < I) GO TO 1      --+
      J = 1
1x    B(I) = B(I) + A(I-1)     <-+
      J = J + 1                  |
      IF (J < 1) GO TO 1x      --+
10    CONTINUE
```

In the following example no such manipulation is possible because of the B(I) dependency in the IF statement:

```
      DO   10  I
1     A(I) = A(I) + 1
      B(I) = B(I) + A(I-1)
      IF  (B(I) < C)  GO TO 1
10    CONTINUE
```

At present, the proposed procedure simply declares a cycle executable in parallel on the DO variable or not, depending on the absence or presence of inter-loop dependencies. Some investigation of "cycle splitting" for the general case shows the analysis to be more complex than the first example suggests.

A cycle will be represented in the ordering relations by a single node at which all variable uses dependent on values generated before entry to the cycle and all variable definitions occurring in the cycle are associated. The latter represent the final values of variables on exit from the cycle. It may be that examination of all the ordering relations shows a cycle node to be in a causal chain sequence. In this case, the cycle is always executed sequentially on the DO variable in its place within the sequence.

3. The Optimal Total Ordering: Overwrite Considerations

Thus far, the requirements laid down for ordering a DO loop to permit parallel execution have excluded consideration of overwrite, that is, the redefinition of a variable or array occurring before all uses of the previous definition have taken place. Preventing overwrite is considered a secondary requirement in ordering the loop because, if need be, it can be handled by using temporary storage. However, this incurs a cost in space and, in some cases, also in time*.

The ordering relations discussed in the last section define a partial ordering of the operations in a DO loop which implies some set of total orderings. The problem then is to choose the ordering that minimizes overwrite (according to some criteria). In cases where there are relatively few orderings to choose from, any ad hoc solution will probably suffice. The general case, however, may involve infinite combinatorics and apparently no general solution to this problem has been found. Heuristic solutions will be investigated based on experience with the I/O latency problem.

---

* Extra code may be needed to restore values to permanent arrays on exit from the DO loop. It may also be needed when overwrite results from re-ordering the original definitions in a loop, causing incorrect exit values.

PART 2: DATA DEPENDENCY IN NESTED DO LOOPS

With regard to the general problem of extending the present methods of analysis to comprehend cases involving multi-dimensioned arrays and nested DO loops: Consider a restricted situation in which we have two tightly nested loops ("tightly" meaning here that all code contained within the outer loop is contained within the inner), the outer on I and the inner on J and both using an increment value of +1, and references to a two-dimensioned array A, in all of which the first index is of the form I+c, the second of the form J+k, where c and k are integers. Assume that there are no control transfer statements.

For any given statement in the loops containing a right-side reference to A, say statement x, we wish to determine whether there are any ordering constraints on x for its SIM execution on 1) J, 2) I, and 3) I and J together. More precisely, we are interested in the ordering of statements necessary in transforming the nest of simple DO loops into any of the following three nests: 1) DO SEQ I/DO SIM J, 2) DO SIM I/DO SEQ J, and 3) DO SIM I/DO SIM J. (For the present, we shall not be concerned with overwrite problems -- "ordering constraints" in this context will refer simply to those relationships between statements necessary in order that the values of an array be generated before they are used.) The following discussion, for the present purpose of clarity, makes no use of the p-graph concept and terminology.

The search for ordering constraints involves, as before, examining the rest of the loop code for dependency relations, to see if the values required for the reference to A in x are generated elsewhere in the nest. But whereas in the case of single loops there were essentially only two kinds of data dependency relations, referred to as " intra-loop" and "inter-loop" dependencies, which were of more or less equal significance in transforming the code to permit exploitation of ILLIAC-type parallelism, a nest of two loops introduces a great deal more complexity. It is no longer true, for example, that a data dependency relationship necessarily implies an ordering constraint. Specifically, there are three distinct kinds of data dependency relations possible, one of which has three different varieties; each of this total of five types has slightly different implications for the transformation of the code; and one reference to an array can be dependent on any number of other statements in various of these ways.

The most straightforward situation is one involving what we shall call simply an " intra-loop" dependency relation, which is precisely analogous to the like-named relation in single-loop code. For example, suppose that the following statements occur in a nested loop of the sort under consideration:

```
y  A(I,J)  = B(I,J) + C(J)
:
x  D(I,J)  =  A(I,J) + 1
```

Assuming that no other statement with a left-side term A(I,J) intervenes, every value used by the reference to A in statement x is generated by statement y during the same iteration of the loop code. In such a situation, all that is necessary (so far as this particular reference to A is concerned) for the DO nest to be transformed into any of the three nests described above is that statement y precede statement x.

A slightly more complicated but still fairly straightforward situation is one involving what we shall call an "intra/inter-loop" dependency relation, that is, where values are generated and used within a single iteration of the outer loop but in different iterations of the inner loop; for example:

```
y  A(I,J+1)  = B(I,J) + C(J)
:
x  D(I,J)  =  A(I,J)
```

When, say, I=1 and J=1, statement y generates a value for A(1,2); when J is incremented and the code executed again, statement x uses this value. More generally, if we represent the reference to A in statement x by $A(f_1, f_2)$, then this kind of dependency relation can occur only if there is at least one statement y with left-side term $A(g_1, g_2)$ such that $g_1=f_1$--otherwise there could be no interaction between the two for a single value of I--and $g_2>f_2$--otherwise x would use a value for any particular element of A before y could generate one. If there is more than one such reference to A, the particular statement generating the values used in x can be determined by examining these references with respect to the second indices alone by a procedure essentially identical to that described in an earlier report for singly-dimensioned arrays in single loops. The statement, say y, thus located must precede x in any transformation of the DO nest to the first or third type of the SIM nests listed above; however, for the second type, involving SIM execution on I alone, this dependency relation requires no ordering constraints, since the sequential execution with respect to J will automatically ensure that generation precede use.

The situations considered thus far are not in principle different from those encountered in the discussion of singly-dimensioned arrays in single loops. At this point in the analysis, however, the consequences of the nestedness begin to make themselves felt. Suppose, for example, that a reference $A(f_1, f_2)$ in statement x was found to use values generated by statement y with left-side term $A(g_1=f_1, g_2)$ in an earlier iteration of the inner loop but during the same iteration of the outer loop--that is, an intra/inter-loop dependency exists between x and y. As in the case of single loops, the fact that the dependency is inter-loop with respect to J implies that during at least one iteration of the J loop (specifically, during (g_2-f_2) iterations) statement x will use an "initial value" of A; but whereas "initial" in the former case meant that the value was generated prior to entry into the loop, here it means simply that the value was generated before the present execution of the J loop was initiated--it may or may not have been generated by a statement other than y during an earlier execution of the J loop, that is, during a previous iteration of the I loop.

Even if an intra/inter-loop dependency is discovered, then, the search for generating statements must continue. (This, of course, is obviously not the case for a simple intra-loop dependency). Only statements containing left-side instances of A with first indices larger than that of the reference to A in x are candidates; those with smaller first indices clearly could not generate values used in x in earlier iterations of the I loop. Disregarding for the moment the conditions imposed by the existence of a finite test value for the DO loops, it should be clear that all of these definition statements will generate values for some of the elements of the array referenced in x prior to that reference--what must be determined is which statement is the last to do so for any given element.

If two left-side terms have different first indices, the one with the larger index will generate a value for a particular element of A during an earlier iteration of the outer loop than the other; if two terms have identical first indices and different second indices, the one with the larger second index will generate such a value during the same iteration of the outer loop but during an earlier iteration of the inner loop; and finally, if both indices are identical, the original ordering of the statements determines the priority of value generation. Tentatively, then, the statement we seek would be a member of the set of the candidate statements with the lowest first index in the left-side term, and, of these, the one with the smallest second index, and,

if there are more than one of these, the one occurring latest in the loop code.

If an intra/inter-loop dependency had previously been discovered, involving a definition of $A(g_1, g_2)$ in statement $\underline{y}$, then of course a further restriction is that the second index be smaller than g_2--otherwise statement $\underline{y}$ overwrites all the values before they can be used in $\underline{x}$. If the statement selected by the procedure just described fails to meet this requirement, all statements with left-side instances of A with the same first index are excluded from consideration (since they too would necessarily be overwritten by $\underline{y}$) and the search begun again with the next greater first index. Suppose that, eventually, statement $\underline{z}$, with left-side term $A(h_1, h_2)$ is selected. If $h_2 \leq f_2$, then $\underline{z}$ will generate values for all those references to A in $\underline{x}$ (after the first (h_1-f_1) iterations of the outer loop) for which $\underline{y}$ fails to generate values. If, however, $h_2 > f_2$, then there will remain at least one iteration of the J loop (to be precise, (h_2-f_2) iterations, again after the first (h_1-f_1) iterations of the outer loop) for which $\underline{x}$ will use an "initial value" for the A reference. Consequently, the search must be continued. Clearly any statements with a left-side reference to A with a first index equal to h_1 are excluded; further, there is now a restriction on the second index, namely that it be smaller than h_2. The search continues in this manner and terminates in one of two ways: either no A definition statements remain for consideration, or sufficient generating statements have been found to produce the greatest possible number of values for the reference to A in $\underline{x}$.

The preceding discussion, however, omitted any consideration of the consequences of particular values for the DO loop parameters. The procedure described above must be modified in certain ways to take these into account. Consider, to begin with, the initial and test values of the outer loop; call them α_I and β_I. The value $R_I = \beta_I - \alpha_I + 1$ represents the number of iterations of the outer loop that occur during a single execution. If the difference between the first index of a left-side instance of A and f_1 is greater than or equal to R_I, then any interaction between the two references is precluded; the search described above, then, ceases when such statements are the only ones remaining to be examined.

Now consider the corresponding parameters in the inner loop, α_J, β_J, and $R_J = \beta_J - \alpha_J + 1$. These have an analogous effect: no statement with a

left-side reference A(p,q) can generate values used in $\underline{x}$ if $(q-f_2) \geq R_J$ (in both inter-and intra/inner-loop dependencies). Additionally, no statement need be considered if $(f_2-q) \geq R_J$. The former case, of course, occurs when q is too much larger than f_2, the latter when it is too much smaller. These restrictions may be restated as limits on the value of q: it must be smaller than an upper limit $L_J = R_J + f_2$ and larger than a lower limit $L_{J1} = f_2 - R_J$. These limits are modified in the course of the search by the discovery of generating statements. Suppose a statement $\underline{y}$ with left-side term $A(g_1, g_2)$ is determined to generate values used in $\underline{x}$. If $g_2 = f_2$, $\underline{y}$ will generate values for all references to A in $\underline{x}$ after the first $(g_1 - f_1)$ iterations of the outer loop; and since no statement remaining to be examined can have a smaller first index, the search is terminated. If $g_2 > f_2$, (as it must be in an intra/inter-loop dependency, and may be otherwise), then L_{J2} is set to g_2, since any remaining statement with a second index higher than g_2 will fail to generate any values that could be used by the reference to A in $\underline{x}$ other than ones that will be overwritten by $\underline{y}$. If $g_2 < f_2$, L_{J1} is set to g_2 for similar reasons.

These limiting values provide one of the direct tests for terminating the search for generating statements: If $(L_{J2} - L_{J1}) \leq R_J$, then there exist no statements remaining to be examined that could generate values used by the reference to A in $\underline{x}$. (The satisfaction of this condition means, loosely speaking, that two generating statements have been found, one with a left-term second index smaller than f_2, the other with a left-term second index larger than f_2, that are "close" enough to "overlap"--that is, f_2 takes no value in the loop that is not taken by the second index of one or the other of the two generating statements.)

An additional consequence of the finiteness of β_J that might be noted here is that several generating statements may have identical first indices; for example:

```
y   A(I+1,J)   = B(I,J) + C(J)
:
z   A(I+1, J+2) = D(I,J) + C(J)
:
x   E(I,J)      = A(I,J+4) + 1
```

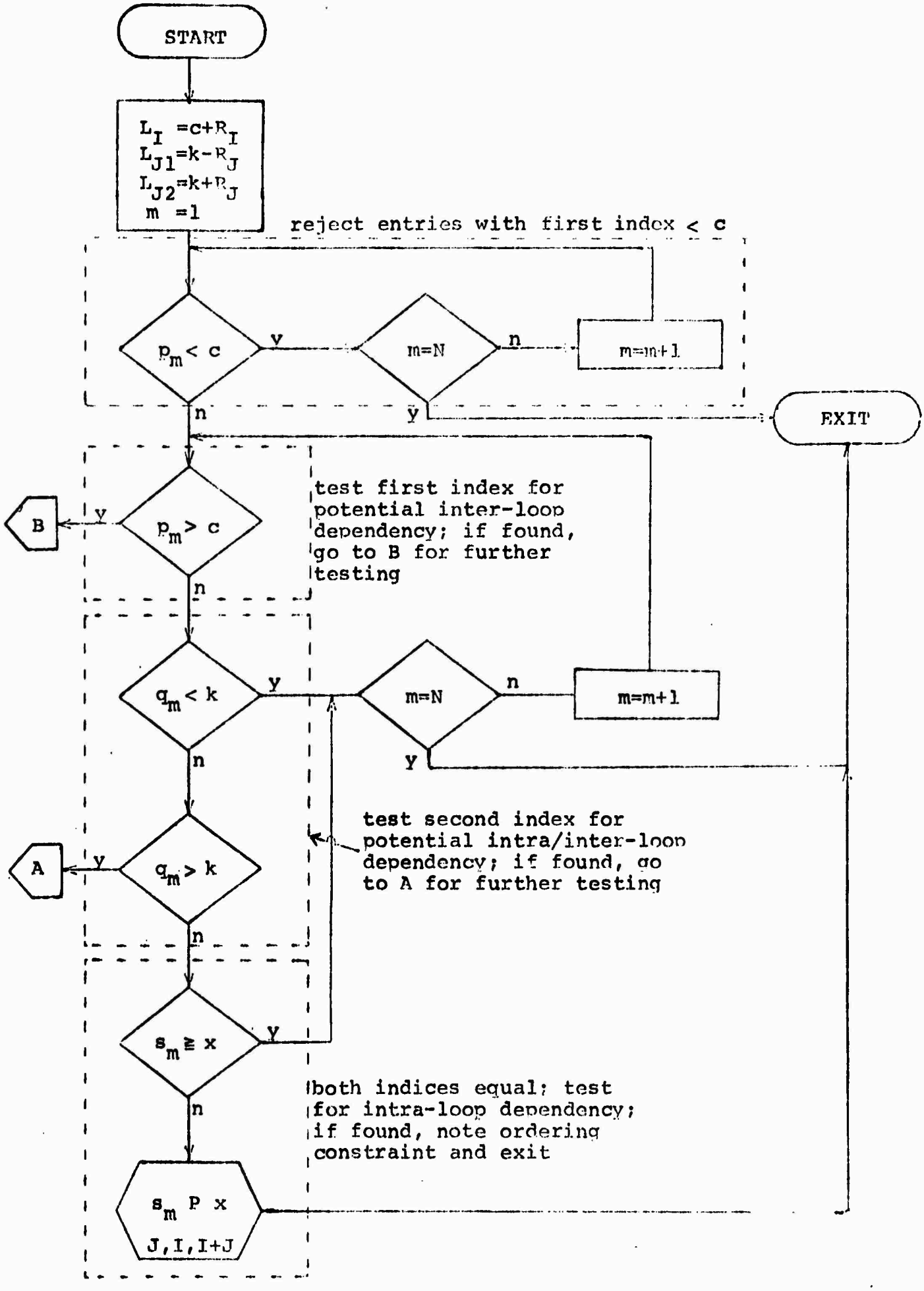
If J ranges from 1 to 10, then y will generate values for use in x for A(I,5) to A(I,10), and z for A(I,11) and A(I,12), for all I except a_1. For any given first index, however, there is at most one generating statement with a second index equal to or greater than i_2.

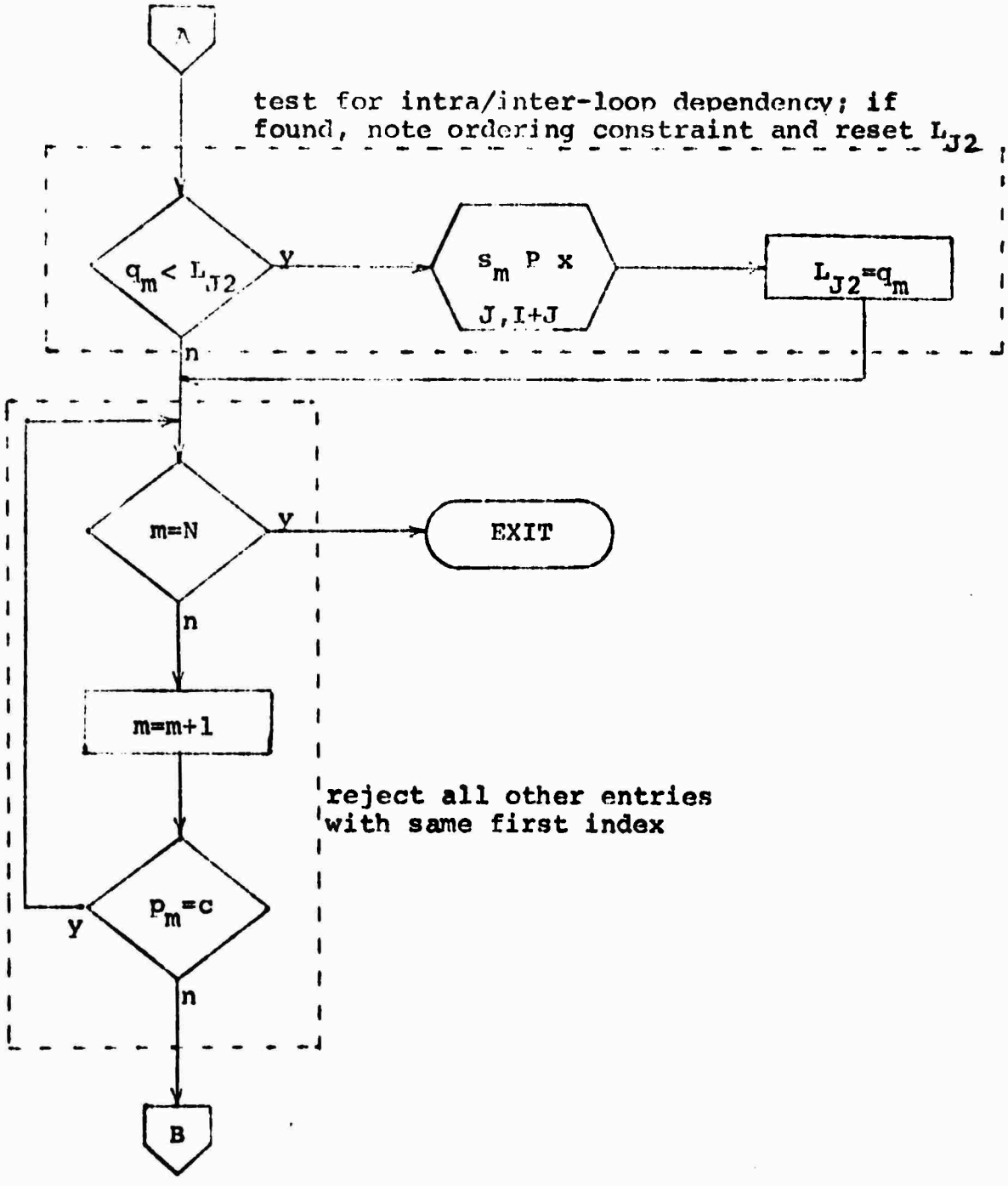
All of the cases discussed above where x is found to depend on a statement with a larger first index may be thought of as simply "inter-loop" dependencies. The three possible varieties of inter-loop dependency, however, have rather different implications for the reordering of statements in effecting a transformation of the DO loops into a DO SIM nest. In no case is there any ordering constraint for the transformation into the first type of nest, involving SIM execution on J alone, since sequential execution with respect to I will ensure that generation precede use. In every case the generating statement must precede x for the third type of nest, the SIM/SIM nest. It is for the second type of nest, involving SIM execution on I alone, that the consequences differ: If the second index is equal to f_2, then the generating statement must precede x; if it is greater than f_2 there is no constraint; while if it less than f_2, SIM execution on I alone cannot be effected.

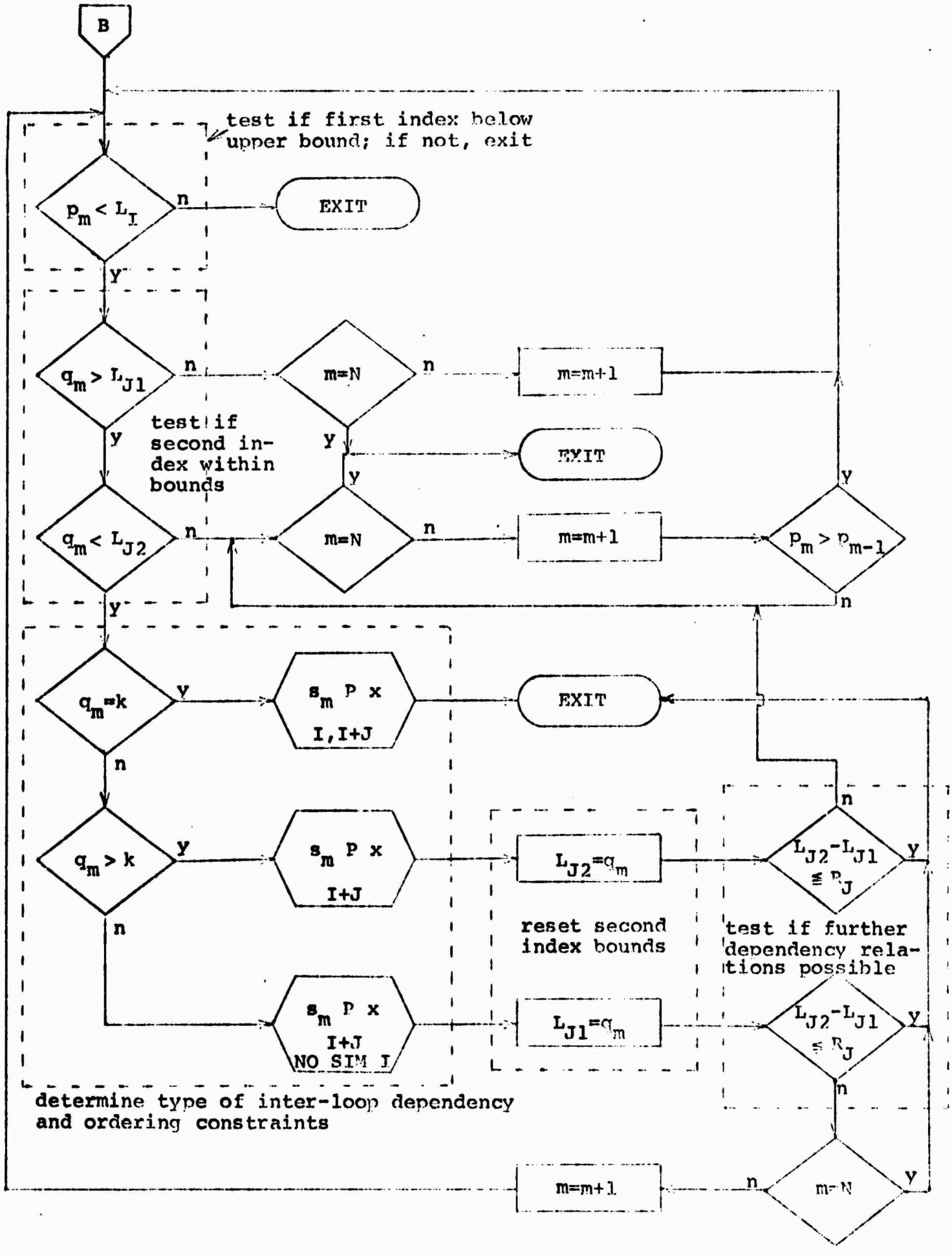
The following flowchart represents a precise statement of the procedure described in the preceding pages. It is assumed that all N left-side instances of A in the body of the loop have been located, and that they have been listed in a table, along with associated statement numbers (which have been assigned sequentially in the original order), in order of increasing indices, the second index varying more rapidly, and, where pairs of indices are identical, in decreasing order of statement number. In the flowchart, the symbols c and k represent the first and second index constant modifiers of the reference to A in x; the symbols s_m, p_m, and q_m represent, respectively, the statement number and the first and second index constant modifiers of the m^{th} entry in the table. (Note that since it is the difference between indices that determine the decisions in searching for data dependency relationships, it is only the constant modifiers that are significant.) An entry is initially examined on the first page of the flowchart; if it is potentially involved in an intra-loop dependency, it is accepted or rejected on that page; if it is potentially involved in an intra/inter-loop dependency, it is tested on the second page; and if it is potentially involved in an inter-

loop dependency, it is tested on the third page. Hexagonal boxes contain the results: "s_m P x" means that statement s_m must precede x in the transformed code, for SIM execution on the loop variables listed at the bottom of the box (where I+J refers to SIM execution on both variables together).

The flowchart should not be taken to be more than a precise summary of the material presented in the test. First of all, it is obviously valid only for nested loops of the restricted sort described at the outset of this section. Secondly, the final form of the algorithm for detecting data dependency in nested loops will probably be closer in spirit to the one described in Appendix A for single loops. Finally, the algorithm inherent in this flowchart does not necessarily embody the basic strategy to be pursued in searching for ILLIAC-exploitable parallelism. For example, rather than simply determining all possible ordering constraints arising from data dependency relationships, it might be better to search for only the kinds of data dependency relationships relevant to each of the possible transformations of the nest, considered sequentially in order of their predetermined desirability (which is related, for example, to the ranges of the DO variables), halting as soon as a particular transformation is determined to be possible.


START
$L_I = c+R_I$
$L_{J1} = k-R_J$
$L_{J2} = k+R_J$
m = 1
reject entries with first index < c
$p_m < c$
y
m=N
n
m=m+1
n
y
EXIT
test first index for potential inter-loop dependency; if found, go to B for further testing
B
y
$p_m > c$
n
$q_m < k$
y
m=N
n
m=m+1
n
y
test second index for potential intra/inter-loop dependency; if found, go to A for further testing
A
y
$q_m > k$
n
$s_m \geq x$
y
both indices equal; test for intra-loop dependency; if found, note ordering constraint and exit
n
s_m P x
J, I, I+J


A
test for intra/inter-loop dependency; if found, note ordering constraint and reset L_{J2}
$q_m < L_{J2}$
y
s_m P x
J, I+J
$L_{J2}=q_m$
n
m=N
y
EXIT
n
m=m+1
reject all other entries with same first index
p_m=c
y
n
B


B
test if first index below upper bound; if not, exit
$p_m < L_I$
n
EXIT
y
$q_m > L_{J1}$
n
m=N
n
m=m+1
y
test if second index within bounds
y
EXIT
y
$q_m < L_{J2}$
n
m=N
n
m=m+1
$p_m > p_{m-1}$
y
n
y
$q_m = k$
y
s_m P x I, I+J
EXIT
n
$q_m > k$
y
s_m P x I+J
$L_{J2} = q_m$
n
$L_{J2} - L_{J1} \leq R_J$
y
reset second index bounds
test if further dependency relations possible
n
s_m P x I+J NO SIM J
$L_{J1} = q_m$
$L_{J2} - L_{J1} \leq R_J$
y
n
determine type of inter-loop dependency and ordering constraints
m=m+1
n
m=N
y

III. ILLIAC I/O OPTIMIZATION

The following section is concerned with minimizing I/O requests for large arrays. Because of disk latency, this effort is essential to effective use of the ILLIAC IV. The scope of the approach is limited to cases where array references in a small number of contiguous statements require more space than is available in core. Attempts are made to re-order these statements such that like array references appear in adjacent statements.

Results have been disappointing and future effort will be directed at leaving statement order intact and calling I/O as early as possible. By partitioning arrays according to which program nodes reference them (as suggested by T. C. Lowe [4]),and then reducing the graph of the program until partition size exceeds core size, it may be possible to locate essential I/O calls. Once it is determined where I/O calls are essential and what arrays these calls reference, the problem of relocating them is similar to removing invariant calculation from DO loops, a technique which already exists in the literature [1].

I/O ORIENTED STATEMENT PERMUTER

I. Purpose: The ratio of disk-seek time to memory time is approximately 4 orders of magnitude on ILLIAC IV. For this reason it is advantageous to issue I/O calls as early as possible, hopefully minimizing time spent waiting for material from disk. It is possible (by rearranging statements while preserving data dependencies) to maximize the amount of I/O which can be backgrounded. The purpose of this effort has been to examine procedures for accomplishing such a rearrangement that will entail as little cost in combinatorics as possible. The problem may be stated formally as: Given a parial order on a finite set and a cost function associated with each linear order on the set, find the linear order of minimal cost that is consistent with the partial order.

II. Scope After the rather intensive work described below had been conducted it became apparent that the scope of this effort is somewhat more limited than had been anticipated. The specific technique described deals with I/O problems of a very local nature, and is applicable in situations which do not seem to arise with astonishing frequency. The effort did, however, afford the investigator deep insight into the nature of FORTRAN code and the possibilities for its deformation.

III. The Permutation Generator

This routine has two parts. Part I computes a partial order matrix, using Warshall's algorithm [3] to obtain the transitive closure of a set of relations of the form a .lt. b, where a and b are statement numbers. Part II generates permutations of the statements consistent with the matrix from Part I. In addition, Part II is loaded with switches and heuristics in an effort to find as short a path as possible to the minimal permutation. These heuristics will be described in section VI.

III.1 Part I--the matrix generator

The data dependencies are generated as follows:

(note: .p. is our partial order relation)

L(a)= variable appearing on left of statement a

R(a)= variables appearing on right of statement a

a.p.b iff:

1) $(a < b)$ and $(R(a) \cap L(b) \neq \emptyset)$, or

2) $(a < b)$ and $(L(a) \cap R(b) \neq \emptyset)$, or

3) $(a < b)$ and $(L(a)=L(b))$ and $\exists x \ni b < x < \inf\{y \mid L(y)=L(b), y > b, R(x) \cap L(b) \neq \emptyset\}$.

Condition (3) is actually too strong, for it defines not a .p. relation but an anti-contiguity relation. If (3) is met, then the restriction on a is that it may not appear between b and any statement whose right side references the "b" activation of variable L(b). In the test program, this condition was overlooked entirely, but it has had little effect on the results and in no way invalidates them.

After the data dependencies are computed, Warshall's algorithm is applied to obtain the transitive closure--an upper triangular boolean matrix whose (i,j)th entry reflects the truth value of the statement "i.p.j".

III.2 The basic permutation generator

The permutation generator is driven by a mask matrix and auxiliary tables, all computed from the closed partial order matrix.

III.2.1 The mask matrix (MSK1)

This is a boolean n x n matrix, where n is the number of statements being permuted. Each column is a position in the final linear order, The (i,j)th entry is 0 if a placement of statement i in position j is legal,

-1 otherwise. The initial setting of the matrix is determined by the NP and NF tables below.

III.2.2 The auxiliary tables

1. IRES: an n-vector which contains the final linear order.

2. JROW, JCOL: n-vectors containing the number of zeroes in a given row (column) of MSK1.

3. NP, NF: n-vectors containing the number of statements which must precede (follow) a given statement. Row I of MSK1 originally contains NP(I) -1's, followed by n-NP(I)-NF(I) 0's,followed by NF(I) -1's. In words, a statement which must precede (follow) NF(I) statements (NP(I) statements) cannot appear in any of the last (first) NF(I) (NP(I)) positions.

III.2.3 The algorithm

This is a stack algorithm which places statement after statement into IRES until either all statements have been placed or an inconsistency has been detected. Subsequent work has indicated that there may be a slightly more efficient algorithm (see Appendix B)

1. L = 1 (set stack depth).

2. Restore MSK1 and tables to (L-1) state (0state is original state).

3. LL=IORD(LL) (statements are handled in a specific order, discussed below).

4. IPERM(LL)=IPERM(LL+1)(move to next permutation at this level).

5. IPERM(LL) .gt.JROW(LL)? (NO, go to 7).

6. Yes, done with this level, pop stack
   L=1? (YES, done).
   L=L-1
   go to 2

7. Place LL in IPERM(LL)th open position in LLth row of MSK1.

8. Mask out row and column of MSK1 taken (JCOL,JROW=1).

9. Mask out all open positions before(after) taken position in rows of statements which must follow(precede) statement LL. Also adjust JROW, JCOL.

10. Is any JROW or JCOL = 0? (YES means as inconsistency, goto 2)

11. Is any JROW or JCOL-1? (YES means we have a forced entry; NO, goto12)

12. Any more to do? (NO, done; YES, push stack: L=L+1, goto3).

There are some obvious bookkeeping details which have been omitted, but in essence the above statement of the algorithm is correct.

We mentioned that the statements are done in a specific order. The reason for this is that it is desirable to minimize the number of non-hits (yeses at 10). To do this, we do statements in increasing order of number of original open spaces (0's) in MSK1. Thus errors are less likely to occur, since as restrictions (i.e. the number of masked-out rows and columns) increase we are dealing with elements which had more open spaces to start with, and so can "stand" to have some crossed out.

IV. The cost function

The cost function has been designed to be the simplest non-trivial I/O simulator possible. Given a permutation it computes total and critical (i.e spent waiting) I/O time. Effort is made to background I/O, but the analysis is not necessarily the most sophisticated possible.

V. The raw result.

For sets of statements of order 7-8 or fewer, assuming that the number of order relations is not impossibly small, it is feasible to examine every linear order to determine the one with minimal cost. For larger sets the combinatoric nature of the problem asserts itself, and heuristics must be applied. In general it is possible to apply heuristics to sets of order 12 or lower, so larger sets are chopped into units of order 10-12.

VI. Heuristic approaches

1. Giving up after a certain number of legal permutations have been found. This method assumes that if an approach (i.e. some other heuristic) is good it will generate low-cost orderings quickly, and thus if no such orderings are found early it is safe to quit. This method is used in conjunction with method 2 below.

2. Generating only those permutations which have all n references to at least one variable in n+1 contiguous statements (n contiguous statements was found to be much too restrictive, and n+2 or greater is too lax). This approach does well, as it should. The reasoning behind it is that any good permutation must have at least one array resident in core for a while (to minimize critical I/O) so there will be more time for backgrounding. The problems with this approach are first

that there is bound to be some duplication of effort between segments dealing with different variables (the same permutation may be a hit for more than one variable), and second that it is not always clear which variables should be made contiguous with respect to references. The first problem is effectively unavoidable, but does not appear to be too serious, for when it arises it means that the permutation we are looking at is probably good (after all, if one contiguity is good, how bad can two be?), and we may find some better ones nearby, as it is an observed fact that good permutations tend to cluster (be generated close to one another).

The second problem is more difficult. Generating permutations based on each variable eats up much of the time advantage of the method. With all variables being examined(the test program uses 12 variables, though it is wildly unlikely that more than 6 or 7 of them will be arrangeable in the required contiguity, and the program recognizes such cases quickly), this method produces slightly better results in slightly shorter time than does the raw method (no heuristics except to give up after finding, say, 1000 legal permutations). There do appear to be some reasonable heuristics to determine which 4 or 5 variables are likely to be best, but these have not been looked into too closely.

The cutoff procedure for this method is interesting, for it has effected huge time cuts with no noticeable loss in power. Examination of permutations based on a variable is terminated if no improvement is found in the first 25 legal permutations, or if the only improvements in the first hundred were in the first ten, or if 125 permutations are examined with no improvement. In each case improvement means improvement over the previous best permutation, where the first permutation is the original linear order. The rationalizations for the three cutoff points are: 1) good variables are good early; 2) variables tend to be characterized by the first few permutations they generate; 3) No cases have been observed where sparse permutations were especially good, and even some of the best sparse ones have been improved on by other variables.

3. Generating only a very small class of permutations, but doing a fair bit of analysis to parametrize the class. The analysis consists of the following steps:

1. Create a table of all references to all variables (i.e. a list of statements for each variable).
2. Determine which pairs of statements are "good" in the sense of having many variables in common and also being able to be made contiguous.
3. Require as many disjoint pairs as possible to be contiguous.

There is evidence that this method is the best of all, effecting a time cut of from 1 to 2 orders of magnitude over (2), and with possibly better results. It is based on the theory that instead of searching a large set of permutations, we will attempt to generate only permutations that are close to minimal. It seems clear that the statements paired are the very essence of a minimal permutation, and in fact may be close to comprising sets of necessary and sufficient conditions for a permutation to be minimal. The one drawback to this method is that unless at least two disjoint pairs are found (and three is much better), there is not enough reduction done to ensure that the restricted class of permutations is small enough, for the idea of this method is to test <u>all</u> permutations which are legal by the above parameters. In about half the cases we cannot find two fairly powerful (two or more variables in common) disjoint pairs. A possible hybrid of methods (2) and (3), would seem to solve the problem, but this has not been tested.

VII. The algorithms were implemented on a PDP-10 computer. The percentage reduction is from original orderings of random statements. The table on the following page summarizes the results of these efforts.

Method	Percentage Reduction (Average)	t(Average CU Time)		
		$n \leq 7$	$8 \leq n \leq 12$	$n \geq 12$
All permutations	Maximum (usually about 50)	$2 \leq t \leq 15$	$t \leq 180$	t proportional to k^n
All permutations (with cutoff applied for $n > 12$)	close to maximum 10-15	$2 \leq t \leq 15$ ---	$t \leq 180$ ---	--- $t \leq 300$
Method 2 (usually about 1/2 of the variables are tested. Analysis might reduce this 1/4 to 1/3 for significant saving)	30-50	---	$t \leq 30$ per variable	---
Method 3 (can only meet conditions 1/2 time)	close to maximum	---	$15 \leq t \leq 30$ (estimated)	---

IV. ILLIAC OVERLAP OPTIMIZATION

The following section is concerned with execution time optimization. Because of the unconventionality of the ILLIAC, techniques are introduced by describing counter examples to usual optimization methods. An attempt is made to define optimization goals and the limits of their effectiveness. A simple optimization algorithm is introduced, and a characterization of the optimization problem is specified.

A Conventional Approach to an Unconventional Machine

Because the number of PE's in the ILLIAC IV is limited to sixty-four, the EXTENDED FORTRAN Compiler maps each SIM assignment statement whose SIM variable is greater than sixty-four into a control loop and an assignment of sixty-four values. The control loop iterates the assignment until it is executed for all values of the SIM variable. More than one SIM assignment statement may occur within a DO SIM loop. Since each statement within the loop is completed before proceeding to the next, an identical control loop is generated for each assignment statement. From a conventional optimization point of view, the repetition of identical loops is time consuming and, in cases where data dependency and overwrite considerations do not interfere, unnecessary. The reduction of identical control loops to a single loop encompassing all the assignment statements in a DO SIM loop, appears to be an effective optimization technique.

Because of overlap between CU and PE execution, the anticipated gain in execution time is negligible. Control instructions are executed in the CU; SIM assignments in the PE's. Since CU and PE execution overlap, unless, in a given SIM assignment, the CU execution time is greater than PE execution time, the elimination of CU instructions will not change combined execution time. An examination of the timing of the instructions we anticipate generating for SIM assignment statements has shown that, except in the simplest cases, combining control loops is an ineffective optimization technique.

Balancing by Allocation

In order to account for execution overlap between CU and PE processing, the concept of code dominance has been developed. We hypothesize that ILLIAC code may be broken into segments in which either the CU or PE instructions take longer to execute. We call the processor which takes longer to execute, and therefore, determines the execution time of that segment, the dominant resource for that segment. In the previous example, SIM assignment statements are PE dominant. ILLIAC optimization efforts must be directed at the dominant resource.

If neither processor is idle over a portion of code, then execution is balanced. Since, in the case of sequential code, it may be possible to execute instructions in either the CU or a single PE, reduction in execution time may be achieved by reassigning instructions from the dominant to the idle resource. This procedure will be referred to as balancing by allocation.

A second example of the unconventionality of ILLIAC optimization: Balancing by Relocation

A "machine independent" optimization technique which has been examined in the literature is the removal of invariant calculation from program loops. In the case of ILLIAC code, significant reduction in execution time may be achieved by moving invariant calculation into program loops. Assume that a programmer has coded a simple assignment statement of the form A=B+C followed by a SIM assignment statement whose SIM variable range is greater than sixty-four. The simple assignment statement is inter-changeable; that is, it may be executed in either the CU or a single PE. The SIM assignment statement generates a control loop and an assignment of sixty-four values. This code is PE dominant. Clearly, execution is reduced by allocating the simple assignment statement to the CU and moving the CU code into the PE dominant loop. (We assume that the difference between PE and CU time within the loop is greater or equal to the CU time necessary to execute A=B+C). Moving code within a processor to a segment where the same

processor is idle will be referred to as balancing by relocation.

***

In summary, execution overlap requires unconventional optimization techniques; allocation and relocation. The first balances by allocating instructions between processors. The second balances by relocating instructions within a processor.

The Indeterminancy of Code Dominance

Execution is a dynamic process. If Maxwell's demon were available, then we could identify, at each moment of execution, the dominant resource. Because of the unavailability of such a device we would like to ascribe the condition of dominance to ILLIAC code rather than the ILLIAC processors. We could then allocate and relocate by means of an algorithm which segments the code such that each segment has a distinct dominance. Unfortunately, this situation does not obtain. Each transfer from a dominant portion of code carries with it a 'surplus' of unexecuted instructions which will effect the dominance of the subsequent portion of code to be executed.

For example, a program block is coded such that a large number of instructions are interchangeable. The block's entry is a merge; one side of the merge is balanced, the other is PE dominant. If execution proceeds from the PE dominant branch of the merge, then the block is optimized by making it CU dominant. If execution proceeds from the balanced branch of the merge, then the block is optimized by balancing it. Resource dominance is both a function of the code being executed and the preceeding code awaiting execution. This condition is somewhat ameliorated by the ILLIAC overlap design, which only queues PE instructions. Consequently, an unexecuted surplus can only occur in the case of PE dominance. Local balancing is a reasonable optimization goal in the sense that it reduces execution time in comparison with executing all interchangeable instructions in a single PE. From a global point of view, a knowledge of 'the most probable path of execution' can make optimization efforts more effective.

Conventional optimization techniques implicitly assume that the fewer instructions executed, the more optimal the code. Techniques which account for overlap do not obey this optimization rule. Nor can this rule be replaced by a local balancing rule. Unfortunately, as this example has shown, it is not sufficient to say that locally balanced code is optimal code.

A Restriction on Balancing ILLIAC Code

A simplifying assumption, namely that transmission time between the two processors is negligible, must be abandoned. A timing asymmetry of significant proportions substantially effects optimization efforts. In general, a load from a single PE to the CU takes twelve times as long as a load from the CU to a PE. There are two ways of approaching this asymmetry.

The first is to restrict the allocation of interchangeable instructions such that PE to CU dependencies (i.e., an operation in the CU utilizes an operand in a PE) do not occur. This is the approach utilized in our allocation algorithm.

A second approach permits PE to CU dependencies, but establishes some minimum number of contiguous CU instructions which must follow the dependency. Our rationale is that the time necessary to load a CU from a PE can be averaged into the overall cost of executing that portion of code in the CU. In the following section it is assumed that inter-processor latency has been accounted for.

Estimating Optimization Effectiveness

Assuming that ILLIAC code can be balanced, it is possible to determine the upper boundary of the optimization effectiveness. That the code can be balanced implies that interchangeable instructions (in this case, sequential code composed of arithmetic statements involving integer addition and subtraction) are available.

Our approach is to assume that a balanced program exists, move the CU interchangeable instructions to a single PE, and compare execution time. (See Appendix C). In brief, allocating and relocating are equally effective, but CU storage optimization (i.e. the utilization of local memory for CU operands) is essential to the balancing effort. The upper bound for a CU optimized balancing effort is 33 per cent reduction in

execution time; if CU memory is not optimized, then the boundary is 15 per cent.

An Inexpensive Algorithm

The optimization algorithm we propose is inexpensive. Its advantages are that it achieves whatever optimization is easily attainable with minimal effort. While only applicable to sequential code, we suspect the approach might be extended to encompass control loops in SIM assignments.

The algorithm is based on the following observations about partitions of macros generated by sequential FORTRAN code. Partition sequential macros into subsets which have no data dependencies with respect to each other. Members of the subsets are linearly ordered. We observe that the execution of any two subsets may overlap. Ideally, all the instructions in one subset would be allocated to the CU, and all the instructions in the other subset would be allocated to a PE. In reality, the CU instruction set is so limited that in many cases, only a portion of the macros in a subset may be executed in the CU. We, therefore, make the following allocation restrictions.

A subset may be entirely allocated to the CU. A subset may be entirely allocated to a PE. A subset may be allocated such that execution begins in the CU and terminates in a PE, in which case the subset will have a single CU to PE dependency.

Observe that if we allocate according to these rules, then there will be no PE to CU dependencies. Now, for any subset, refer to those macros allocated to the CU as the CU portion of that subset, and the macros allocated to a PE as the PE portion of that subset. A subset may have a CU portion, a PE portion, or both.

Observe that for any two subsets, the execution of a CU portion and a PE portion may overlap. The objective of the allocation algorithm is to execute the PE portion of the nth subset, while executing the CU portion of the $n+1$ subset.

We now introduce timing considerations. Although the subsets may be executed in any order, it is desirable to avoid the following condition: the execution of the PE portion of a subset is delayed because the execution of the CU portion of that same subset is not complete.

We therefore, introduce the following ordering restrictions. For each portion of a subset, compute an execution time estimate for the respective processor. Order the subsets such that for each CU to PE dependency, the sum (over all the previous subsets) of the CU time is less than the sum of the PE time. The resulting ordering is: Subsets with PE portions alone first. Subsets with CU portions alone last. Subsets with both portions ascending from maximal PE and minimal CU to minimal PE and maximal CU. A brief description of the algorithm follows.

The subsets correspond to FORTRAN statements which have no data dependencies with respect to one another. Apply a method proposed by Ramamoorthy [2] to partition sequential statements. The macros generated by these statements correspond to the linearly ordered macros which are members of the subsets.

Apply the following allocation rule: Assign the macros in a statement to the CU until an PE dependent macro (i.e., a multiply) is encountered. Assign that instruction and the remainder of that statement to a single PE. Attach an execution time estimate to the portions of each statement. Assign an index to each statement according to the difference between CU and PE estimates. Order the statements according to the value of the index, smallest values first. The resulting order minimizes PE idle time.

In order to take advantage of hardware buffering, care must be taken when issuing code to interleave CU and PE instructions.

A Characterization of a Block Optimizer for EXTENDED FORTRAN

The following characterization is for a block optimizer. In the case of EXTENDED FORTRAN we define a block as a set of statements in sequential order having one entry and multiple exits. Since each statement in a DO SIM loop is executed in sequential order, a block may contain SIM assignment statements, each with an identical control loop. For our present purposes, the cyclical nature of the SIM control loop will not explicitly appear except as a marking.

A block of ILLIAC code may be characterized as a marked tree, with nodes and edges corresponding to operations and operands respectively. Depending on the character of the operation it represents, a node is marked PE dependent, CU dependent, or interchangeable. In addition, the nodes which, in the actual code, are nested in SIM control loops are identified as iterated nodes.

Allocation may be characterized as a reduction procedure applied to the tree. The objective of such a procedure is a tree of CU and PE nodes. While in the previous algorithm, the first PE node encountered consigned the remainder of the statement to a PE, a more extensive examination might reveal that a large number of interchangeable instructions warrant returning calculations to the CU. Consequently, the first reduction combines interchangeable nodes and assigns CU timing estimates to them. The second reduction begins in a CU node and combines CU and interchangeable nodes until a PE node is encountered. PE nodes are combined until an interchangeable node is encountered. If the CU time estimate for the interchangeable node is above some minimum (this is a function of PE to CU latency and is unknown at this point), then assign a new CU node and continue. Otherwise, combine the interchangeable node with the PE node. Continue this reduction until the tree contains no interchangeable nodes.

Relocation may be characterized as a deformation of the reduced tree. While in the previous algorithm, the ordering restrictions were quite simple, in the present case, the determination of orderings appears to be computationally explosive. The nodes must be ordered such that data dependencies are preserved and that timing order correlates to logical order. In other words, for each CU-PE dependency, the sum of CU time is less than the sum of PE time; for each PE-CU dependency, the converse. In addition, care must be taken to keep iterated nodes clustered. A further restriction is that only invariant calculations may be moved into interated clusters.

We suspect that an extension of the first algorithm, i.e. partitioning statements before reducing the graph,will prove to be the most practical optimization approach to ILLIAC code.

APPENDIX A: ALGORITHM FOR DETERMINING DATA DEPENDENCIES

This algorithm is based primarily on the p-graph material presented in Shapiro and Saint's The Representation of Algorithms [5]. The final section, on subscripted variable dependencies, extends the analysis to include DO loops referencing singly-dimensioned arrays, which of course potentially embody parallelism exploitable on the ILLIAC IV.

The main sections of the algorithm are:

1) Analysis of the flow logic and accumulation of variable use statistics

2) Completion of p-graphs, by flow block and by node

3) Detection of inter-DO loop dependencies for subscripted variables.

DETERMINATION OF FLOW BLOCKS

One scan is made over the code to determine the basic flow blocks and, at the same time, record all variable uses within each flow block. It is initially assumed that statements having labels are referenced elsewhere, that is, are the start of a new flow block, in order to eliminate an extra scan. Information is also recorded in terms of "nodes" to provide the skeleton for the final p-graphs: statements are broken down into one or more nodes according to type and additional nodes are assigned to the entry and exit points of each flow block.

The scan records flow data in a Flow Block Table having one entry per flow block. Entry format:

FB#	NODEN	NODEX	LABEL	DEF	REF

FB#: flow block number

NODEN: entry node number

NODEX: exit node number

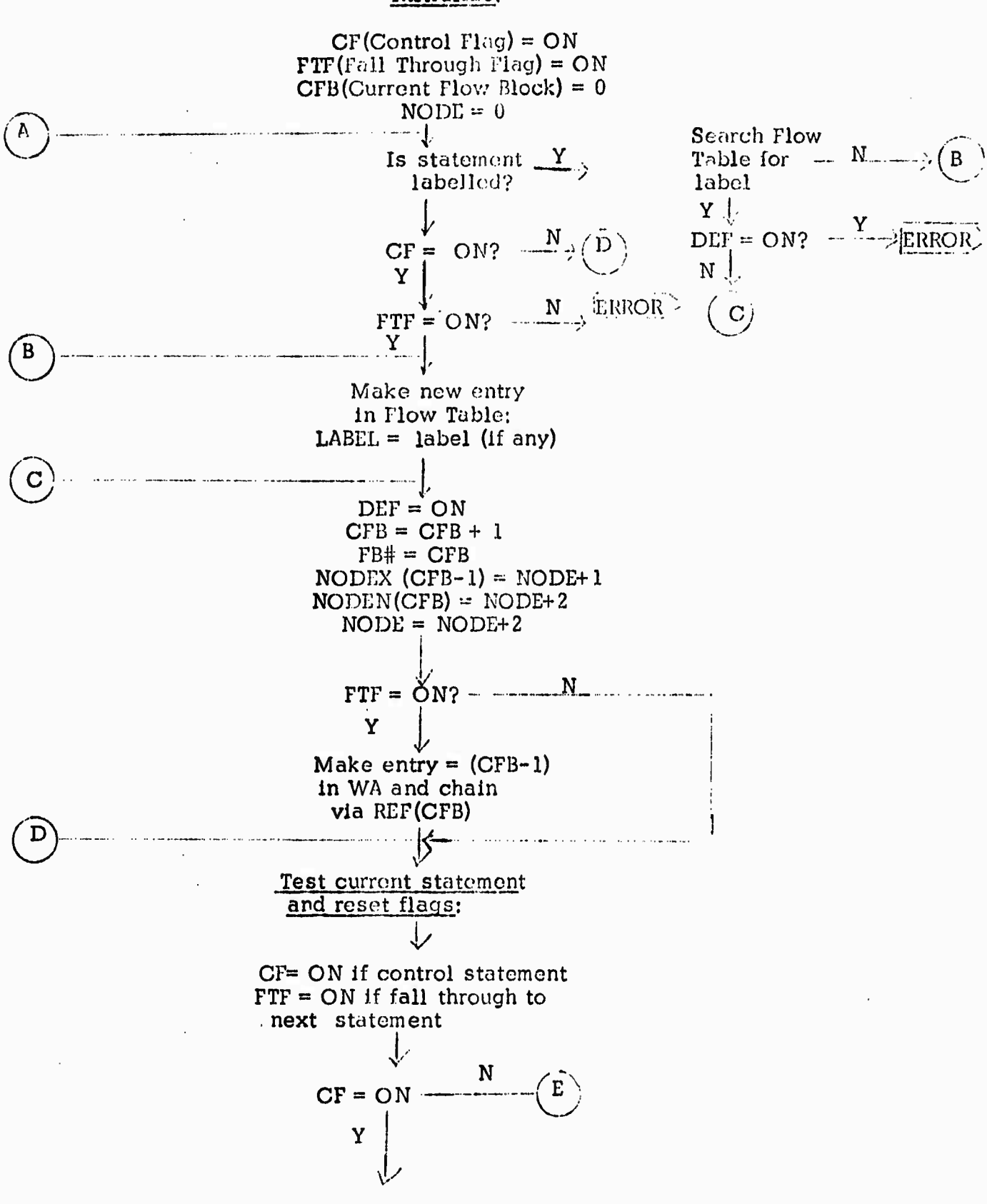
LABEL: label of first statement in block (if any)

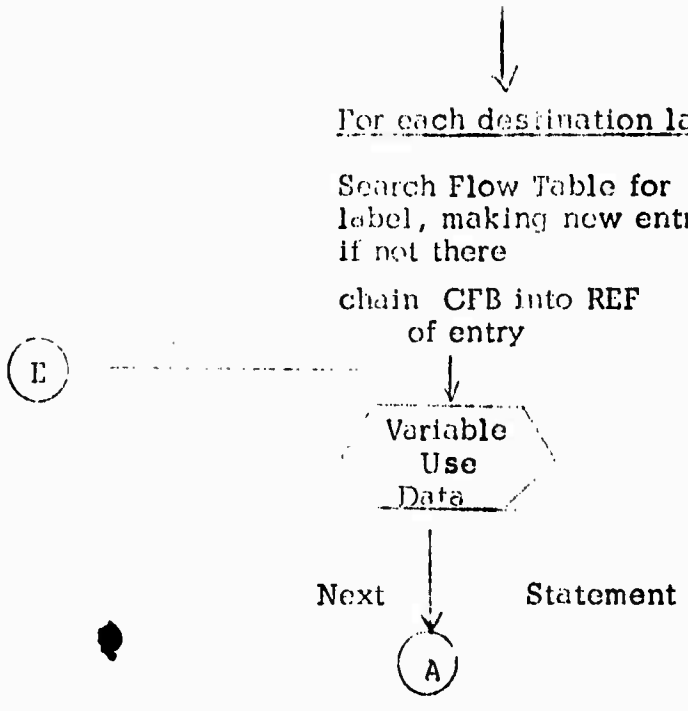
DEF: defined flag = ON if LABEL is the label of some statement; OFF, otherwise,

REF: referenced pointer pointing to the chain of flow blocks which reference (transfer) to this flow block. Zero if none.

A work area WA is used to store the REF chains.

Procedure:


Initialize:
CF(Control Flag) = ON
FTF(Fall Through Flag) = ON
CFB(Current Flow Block) = 0
NODE = 0
A
Is statement labelled?
Y
Search Flow Table for label
N
B
Y
DEF = ON?
Y
ERROR
N
C
CF = ON?
N
D
Y
FTF = ON?
N
ERROR
Y
B
Make new entry in Flow Table:
LABEL = label (if any)
C
DEF = ON
CFB = CFB + 1
FB# = CFB
NODEX (CFB-1) = NODE+1
NODEN(CFB) = NODE+2
NODE = NODE+2
FTF = ON?
N
Y
Make entry = (CFB-1) in WA and chain via REF(CFB)
D
Test current statement and reset flags:
CF= ON if control statement
FTF = ON if fall through to next statement
CF = ON
N
E
Y



The flow block part of the procedure is completed by the construction of the Flow Block Connectivity Matrix FBCM from the completed Flow Block Table. This is a 2-dimensional Boolean matrix giving the direct connections between flow blocks: FBCM(I,J) = 1 if flow block I transfers to flow block J.

To construct the matrix, the Flow Table entries are cycled. For each entry, J = FB# and I = REF(J) chain flow block numbers. FBCM (I,J) = 1 for all such I,J pairs.

The following additional errors may be found:

1) J = 0 => statement label referenced but not defined
2) REF= 0 => labeled statement not accessible
3) FBCM(I,J) = 1 and there is no other non-zero element in row I or column J => flow block J can be appended to flow block I.

   This is the case of generation of an extraneous flow block. It probably does not pay to fix it.

Artificial entry and exit blocks are added to the set of flow blocks. The entry block, numbered 0, precedes all original entry blocks. The exit block, numbered N = (last flow block number + 1), succeeds all original exit blocks.

Variable Use Data

At the same time that a statement is passed through the flow block scan it is also examined for variable uses. The data collected here will furnish a record for each variable of its definitions/references in each flow block and at each node within a block. Redundant information is recorded, that is, by flow block as well as node, since the next procedure - completion of p-graphs - can be more economically executed over flow blocks than individual nodes.

Statements are broken down into nodes in the obvious way: assignment statements (or the assignment part of a logical IF) become 2 nodes - the right hand side (use = referenced) followed by the left hand side (use = defined, except subscripts). The first (or only) node of an IF statement represents the conditional expression (use = referenced). READ/WRITE statements are single nodes (use = defined/referenced).

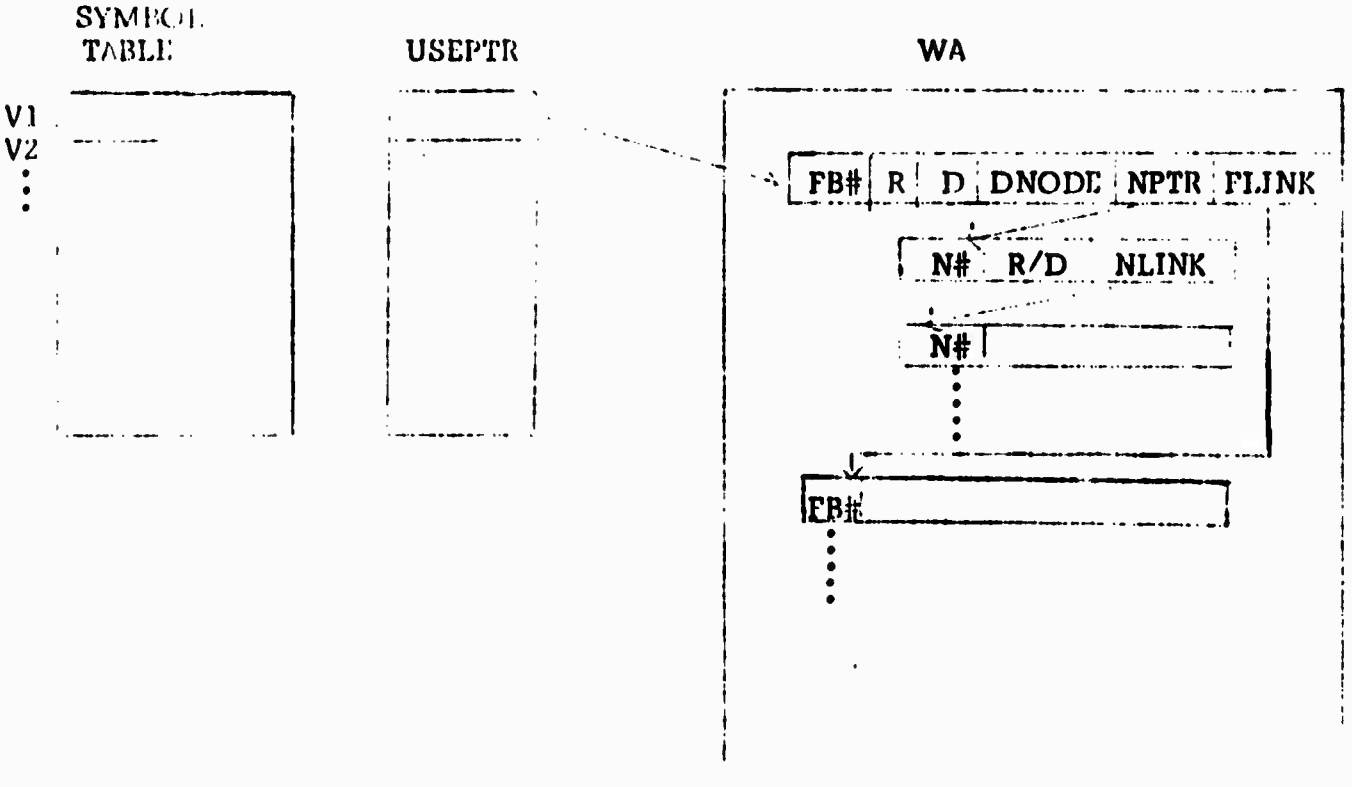
The data may be recorded as follows. Assume that a symbol table for all variables has already been generated and that the variables at this stage are represented by pointers into the symbol table. A new symbol table field USEPTR will be temporarily appended. For each variable, its USEPTR field will contain a pointer to a chain of data entries in the work area WA describing its uses. There are 2 types of data chains constructed in WA. The first, the flow block chain, is located directly by USEPTR. The fields of a flow block entry are:

1) FB#: flow block number.
2) R : referenced flag. ON if variable is referenced in this flow block.
3) D : defined flag analogous to R.
4) D node: node number of last definition of variable in the block.
5) NPTR: pointer to node chain for this flow block.
6) FLINK: ptr to next flow block entry for this variable.

The second type of chain is the node chain which is located by the NPTR of its flow block entry. The node entry fields are:

1) N# : node number
2) R/D: referenced or defined flag
3) NLINK: pointer to next node entry in the chain.

Schematically:



The process of recording the above data is simple. As each statement is passed through the scan, it is broken down into nodes and each node is examined for variable appearances. The USEPTR field of each variable is accessed to locate its current flow block entry in WA. If the FB# of this entry = CFB (current flow block), the R/D/DNODE fields are appropriately updated. If FB# ≠ CFB or USEPTR = 0, a new flow block entry is made and chained in via USEPTR, FB# = CFB and other fields are set appropriately. In both cases, a node entry is also generated.

For all variables, data for the entry and exit blocks is the same: for the entry block, D = ON at its single node DNODE = 1; for the exit block, R = ON at its single node = (last assigned node + 1).

COMPLETION OF THE P-GRAPH

The preceeding procedures need be performed only once over the code to be analyzed; the following procedures must be applied separately to each variable in the code.

As stated earlier, the completion process will be performed over the flow blocks of the program rather than the individual nodes. In general, each flow block will be thought of as having 2 nodes - an entry and an exit node. All entry nodes will initially be reference ("uncircled") nodes. If a definition takes place within the block, the exit node will be circled, referring to the last definition in the block. The result of this procedure will be the determination of the "equivalence class" to which a variable belongs at the entry node of every flow block. (It is then trivial to resolve data dependencies node-wise within flow blocks.) An equivalence class will be defined by the type and place of generation of the class. A class generated by a single definition is designated type "D"; by a merge, type "M". The place of generation is the flow block at the node of definition in the first case, the merge block at its entry node in the second case.

The procedure described here is based on the p-graph algorithm with some modifications. Initially, all originally circled nodes are "propagated" to their successors, but thereafter only new merge nodes are propagated until no more are found. This raises the problem of "zeroing out" only the subset of non-merge nodes affected by a new merge node. However, since it seems desirable anyway to keep track of the set of equivalence classes related to each merge node, this can be done in a way that will record the equivalence class associated with every entry to a merge. Thus a new class always overlays the old class in its assigned slot without destroying any information.

Data Used in Procedure:

ECM - Equivalence Class Matrix. This matrix parallels the FBCM but its values are the names of equivalence classes. The element FBCM (I,J) (if not null) is the equivalence class passed from flow block I to flow block J. The format of an entry is "type/FB#" where type = D or M, FB# = number of flow block where generation of class occurs.

USI - Table - has one entry per flow block. Entry format:

R	D	DNODE	M	NEC/MNODE

This table describes the circled nodes and the equivalence class on entry to each flow block. The R, D, DNODE fields are taken directly from the flow block chain data generated in the last procedure. The M flag is turned ON when a flow block is found to be a merge point of more than 1 equivalence class. The NEC/MNODE describes the entry equivalence class (NEC) for this flow block unless M is ON, when it designates the entry node (MNODE) instead.

MLIST - Merge List. The numbers of merge blocks are placed on this list as they are discovered during the procedure.

SSTACK - Successor Stack. Push down stack used to store the numbers of successor flow blocks of a node during the propogation process. Entry format: PFB, SFB (see below)

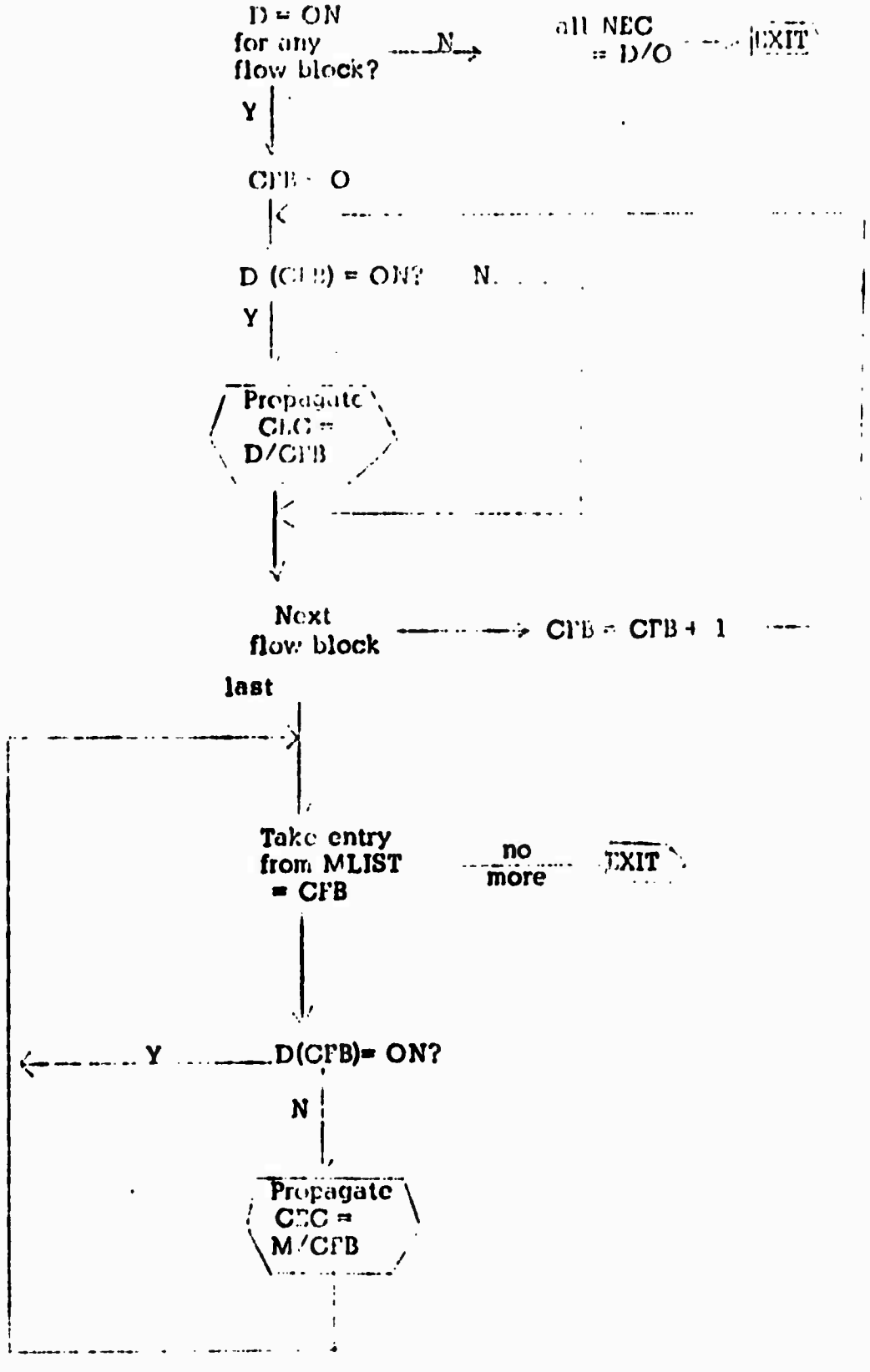
CEC - Current Equivalence Class. Same format as ECM entry.

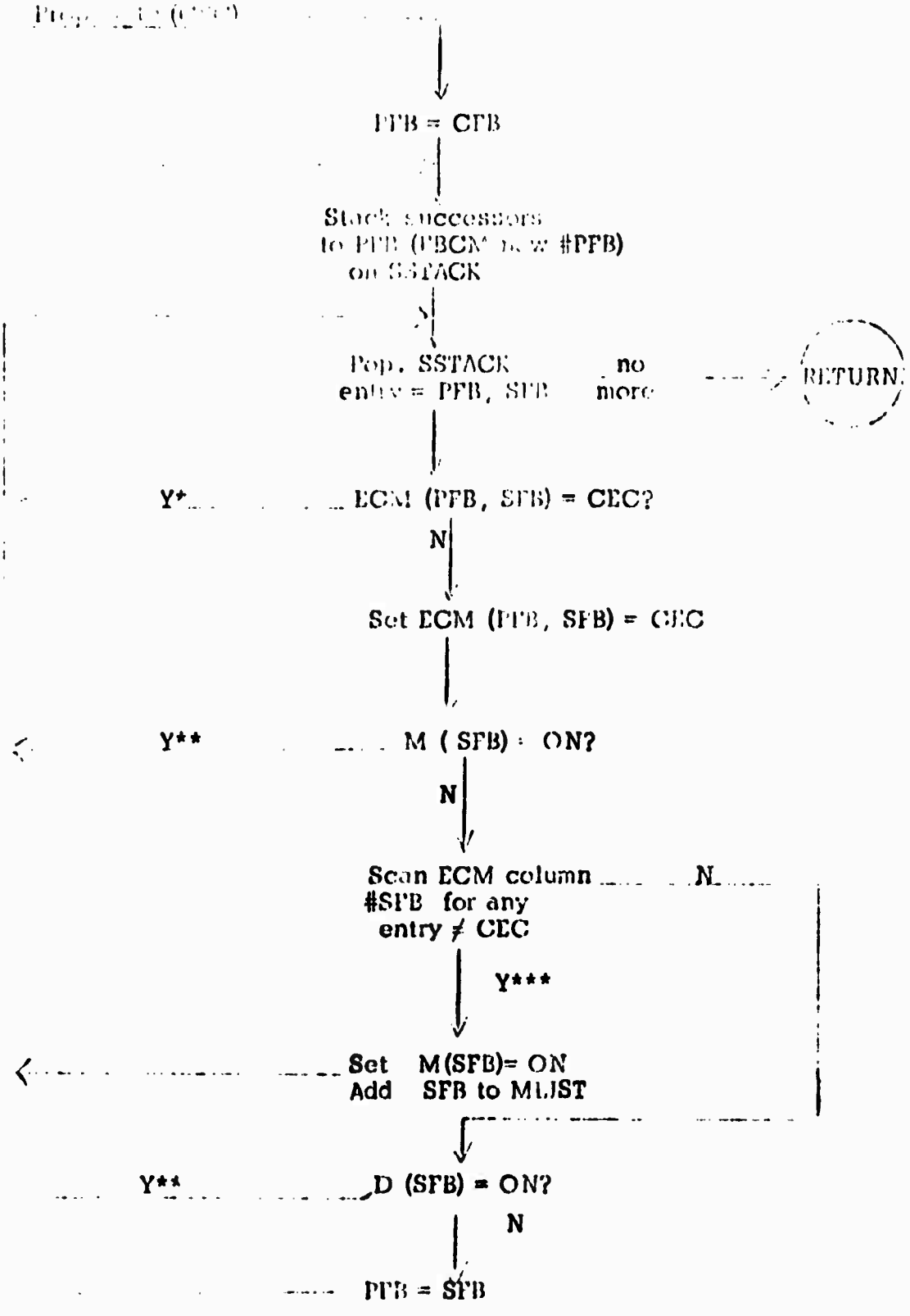
CFB - Current Flow Block.

PFP - Propagating Flow Block.

SFB - Successor Flow Block.

Procedure


D = ON for any flow block?
N
all NEC = D/O
EXIT
Y
CFB = 0
D (CFB) = ON?
N
Y
Propagate CLC = D/CFB
Next flow block
CFB = CFB + 1
last
Take entry from MLIST = CFB
no more
EXIT
Y
D(CFB)= ON?
N
Propagate CLC = M/CFB



* Stop propagation at completion of cycle

** Stop propagation at circled node

*** New merge

The procedure is completed by the filling in of the NEC/MNODE fields of the USE table. If M = OFF for, say, the entry for flow block #J, NEC is obtained by scanning column J for any non-null entry (all are the same). If M = ON, MNODE is set to the entry node of the flow block (NODEN field of the Flow Block Table). Note that the set of merging classes in the latter case is obtainable from the ECM column corresponding to the merge block.

The completed USE Table, together with the FBCM giving the connections between flow blocks, defines the flow block p-graph.

The next procedure passes over the nodes of the flow blocks to make the final assignment of equivalence classes. The information is obtained from the USE Table and the node chains generated in the Variable Use Data section. Three fields will be added to each node entry to record the equivalence class:

FDEF = flow block number of the class
NDEF = node number of the class
TDEF = type of class: D or M

Procedure:

Loop flow block chain in WA located by USEPTR (variable):

If M(FB#) = OFF:

TDEF, FDEF = NEC(FB#)

If TDEF = D, NDEF = DNODE (FDEF)

If TDEF = M, NDEF = MNODE (FDEF)

If M(FB#) = ON:

TDEF = M
FDEF = FB#
NDEF = MNODE(FB#)

Loop node chain located by NPTR(FB#):

If R(N#) = ON, enter current TDEF, FDEF, NDEF in node entry

If D(N#) = ON, reset FDEF = FB#
NDEF = N#
TDEF = D

Next node entry

end of chain

Next flow block entry

end of chain

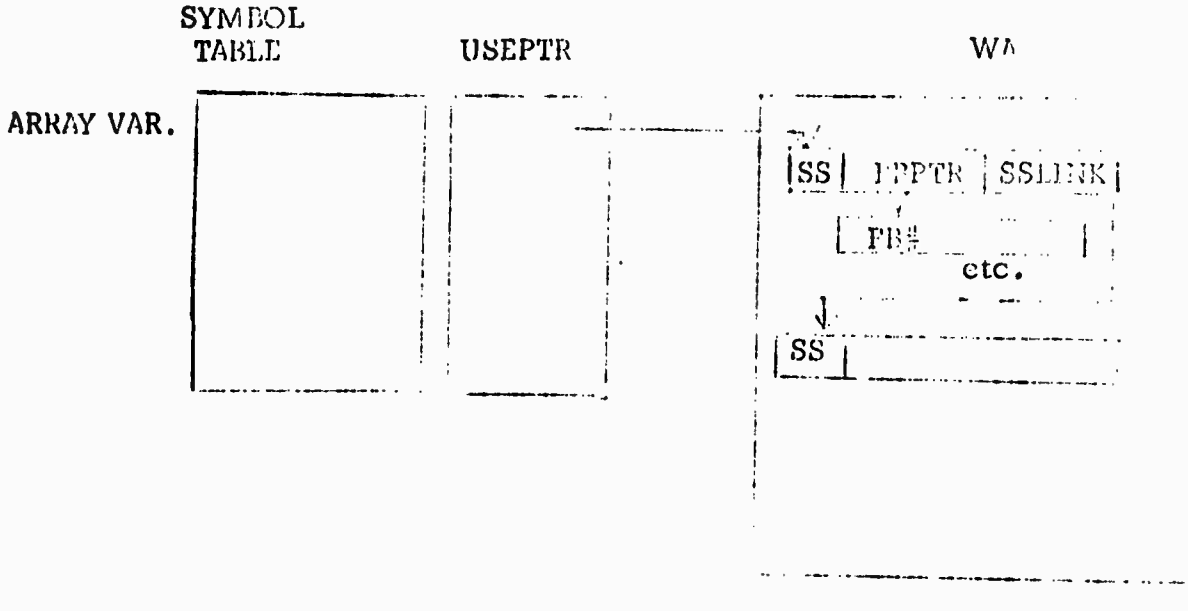
Exit

SUBSCRIPTED VARIABLES

The previous analysis does not distinguish between different elements of the same array in determining data dependencies. Work in this area has been done only within the limited context of the ILLIAC problem, that is, the analysis of DO index subscripted variables in DO loop code for the purpose of finding inter-loop dependencies. The previous procedures can be rather simply modified to find dependencies in this special case:

Variable Use Data

A "subscript" chain is interposed at a level higher than the flow block chain. A subscript entry is made for each unique subscript form associated with an array variable.



Completion of p-Graph

This procedure, through the propagation process, is performed separately for each subscript form of the array variable. In this case, multiple ECM and USE tables must be kept - a set for each subscript form.

The subscript forms are then examined and are assigned indices 1,2, ..., n according to decreasing value of the subscripts. An inter-loop dependency occurs when there is a use of an initial value (equivalence class D/O) on the p-graph corresponding to index $n' \geq 1$ and there is another graph of index $n'' < n'$ showing a non-initial value at the exit block. That is, in this case the value for the D/O use was actually generated in a previous iteration of the loop.

The following procedure updates the data tables to reflect inter-loop dependencies. It is necessary now to append a subscript index field to ECM entries since there will be "cross p-graph" references. (In the initial completion, all indices = the index of the graph). The NEC field of USE entries must be similarly expanded.

(1) I = 1

(2) The NEC/MNODE fields of the USE table for index I are completed. The value (entry equivalence class) for the exit block will re-define all D/O equivalence classes for index (I+1). If this value is itself D/O, skip to step (4).

(3) The ECM corresponding to index (I+1) is scanned and all appearances of D/O are replaced by the above value, setting subscript index = index of this value. In particular, values in the column corresponding to the exit block are now updated.

(4) I = I+1. If $I \leq n$, go back to (2).

APPENDIX B: OBSERVATIONS ON PARTIAL ORDERINGS

Problem: Given a finite set A and an irreflexive partial order $r \subset A \times A$, how many linear orders on A are consistent with r?

Def 1: $r \subset A \times A$ is an irreflexive partial order if

1) $\forall\, a \in A,\ (a,a) \notin r$

2) $(a,b)\ (b,c) \in r$

Note: $(a,b) \in r$ is also written as $a <_r b$ or $a < b$ if r is understood.

Lemma 1: if r_1 and r_2 are (irreflexive) partial orders then $r_1 \cap r_2$ is a partial order.

Proof: 1) $\forall\, a \in A,\ (a,a) \notin r_1 \cap r_2$

2) $(a,b) \in r_1 \cap r_2 \Rightarrow (a,b) \in r_1,\ r_2$

$(b,c) \in r_1 \cap r_2 \Rightarrow (b,c) \in r_1,\ r_2$

thus $(a,c) \in r_1$, and $(a,c) \in r_2$, so $(a,c) \in r_1 \cap r_2$.

Def 2: $L \subset A \times A$ is a linear order if it is an irreflexive partial order and $\forall\, a, b \in A,\ a \neq b \Rightarrow (b,a) \in r$ or $(a,b) \in r$

Linear orders are also called chains.

If $\{a_1, \ldots a_n\} \subset A$ is linearly ordered by $a_1 < a_2 < \ldots < a_n$, then we denote this by $< a_1, a_2, \ldots a_n > \subset r$ [i.e. $< a_1, \ldots a_n > = \{(a_1, a_j) \mid 1 \leq i < j \leq n\}$]

Def 3: The restriction of r to $B \subset A$, written $r|B$, is the set $\{(a,b) \in r | a,b, \in B\}$

Def 4: The primitive of a partial order r, written p(r) is the set $\{(a,b) \in r | \nexists\, c \in A\ (a,c),\ (c,b) \in r\}$.

Note: In Graph Theory the primitive is sometimes called the Hasse diagram.

Def 5: The transitive closure of a subset $Q \subset A \times A$ is defined by:

$C(Q) = \bigcap \{Q' \supset Q \mid Q' \subset A \times A \text{ is a partial order}\}$

It follows immediately from Lemma 1 that C(Q) is either a partial order or all of $A \times A$.

Def 6: If $Q = \{(a_{11},\ldots,a_{1k_1}),\ldots,(a_{n1},\ldots,a_{nk_n})\}$ then the extract of Q, written Q', is defined by

$Q' = \{a_{ij} \mid i = 1,\ldots,n; j = 1,\ldots,k_i\}$

Lemma 2: $C(p(r)) = r$, also, if $X_1 \supset X_2$ then $C(X_1) \supset C(X_2)$

Proof: Let $X_1 \subset A \times A \supset X_2$. If Q is a partial order and $Q \supset X_1$, then $Q \supset X_2$ thus $X_2 \subset C(X_1)$ but by Lemma 1, $C(X)$ is a partial order (unless it is $A \times A$ in which case $C(X_1) \supset C(X_2)$ for any $X_2 \subset A \times A$)

Thus, since $C(X_1)$ is a partial order containing X_2, the intersection of all partial orders containing X_2 is contained in $C(X_1)$. Thus $C(X_2) \subset C(X_1)$.

Now consider r and its primitive p(r). Since $p(r) \subset r, C(p(r)) \subset r$.

Assume the inclusion is proper. Thus $\exists$ (a,b) $\in$ r such that (a,b) $\notin C(p(r))$.

Now $\exists$ c $\ni$ (a,c), (c,b) $\in$ r for otherwise (a,b) $\in p(r) \subset C(p(r))$. Also not both (a,c) and (c,b) may be in p(r) for otherwise any partial order containing p(r) would contain (a,b) by transitivity and (a,b) would be in p(r). Assume without loss of generality that (a,c) $\notin$ p(r). Thus $\exists$ $c_1 \ni (a,c_1)$, $(c_1, c) \in r$. Further $c_1 \notin \{a,b,c\}$ for by transitivity (c,b) $\in$ r and we know that (c_1, c) and (a,c) $\in$ r. At the nth step we have $c_n \notin \{a,b,c,c_1,c_2,\ldots,c_{n-1}\}$.

Thus we will create an infinite sequence of distinct points of A.

But A is finite. Hence the inclusion cannot be proper, and $C(p(r)) = r$.

Lemma 3: For any partial order r there is a linear order L such that $L \subset r$.

Also, if L is a linear order on A and $o(A) = n$, $o(L) = n(n-1)/2$

Proof: Let r be a partial order on A. If r is linear we are done, so assume it is not. Thus $\exists$ a,b $\in$ a $\ni$ (a,b) and (b,a) $\notin$ r. Let $r_1 = C(r \cup \{(a,b)\})$, r_1 is a partial order properly containing r.

Assume r_1 not linear. Thus $\exists$ $a_1, b_1 \in A \ni (a_1,b_1)$ and $(b_1,a_1) \notin r$.

Proceeding in this manner we can construct an infinite ascending sequence of subsets of $A \times A$. Thus for some n, $r_n \supset r$ is a linear order.

Now let $o(A) = n$ and L be a linear order on A. Consider $x \in A$

$\forall y \in \Lambda$, $y \neq x$, $(x,y) \in L$ or $(y,x) \in L$ but not both. Thus each x appears in n-1 elements of L. Thus $o(L) = n(n-1)/j$, where j is the number of times we have counted each pair. But $j = 2$ because (x,y) is counted exactly twice: once as a pair containing x and once as a pair containing y.

Def 7: a) The predecessor set of element $a \in A$, when A is ordered by r, is

$P_r(a) = \{b \in A \mid (b,a) \in p(r)\}$

b) The following set of $a \in A$, A ordered by r, is

$F_r(a) = \{b \in A \mid (a,b) \in p(r)\}$

The predecessor (following) set of a subset X of A is the set of maximal (minimal) elements of $\{y \in A \mid x \in X \Rightarrow y < x\}$ $(\{y \in A \mid x \in X \Rightarrow y > x\})$

Def.8 A merge point of a partial order r is a point a such that:

1) $a \in p(r)'$

2) $o(P_r(a)) \geq 2$ or $o(F_r(a)) \geq 2$ (or both)

If $o(P_r(a)) \geq 2$, a is a left merge point. Similarly, if $o(F_r(a)) \geq 2$, a is a right merge point

Let $\mu(r)$ be the set of merge points of r

Def 9: A chain $<a_1 \ldots a_n> \subset r$ is a base chain of r iff:

1) no a_i is a merge point of r for $1 \leq i \leq n$

2) $(b,a_1) \in p(r) \Rightarrow b \in \mu(r)$

3) $(a_n, c) \in p(r) \Rightarrow c \in \mu(r)$

Def 10: A pair of merge points of r, p_1, p_2), is ready if there are base chains $<a_1,\ldots,a_n>$ and $<b_1 \ldots b_m>$ of r such that $\{(p_1,a_1), (p_1,b_1), (a_n,p_2)$ $(b_m,p_2)\} \subset p(r)$

Lemma 4: If $<a_1,\ldots a_n>$ and $<b_1,\ldots b_m>$ are base chains of r they are equal or disjoint.

Proof: Suppose $a_i \in <a_1,\ldots,a_n>'$. We will show that $<a_1,\ldots,a_n>$ is the only base chain containing a_i. This will clearly be sufficient to prove the Lemma. Thus let $<b_1,\ldots,b_m>$ be another base chain containing a_i and let $a_i = b_j$.

If $i = 1$ then $a_i = b_j$ so $P(a_i) \subset \mu(r)$. Thus $F_r(b_j) \subset \mu(r)$ and $j = 1$ since no member of a base chain is merge point. If $i \neq 1$ then $P_r(a_i) = \{a_{i-1}\}$ and $P_r(b_j) = \{b_{j-1}\}$, so $a_{i-1} = b_{j-1}$ and proceeding by induction $a_1 = b_1$. We can similarly show that for any k, $a_k = b_k$. Thus we are done unless

$m \neq n$. Assume $m > n$ without loss of generality. Since $\Gamma_r(a_n) = \emptyset$ or {a merge point} then $\Gamma_r(b_n) = \emptyset$ or {a merge point}. Thus $b_{n+1} \in \langle b_1, \ldots, b_m \rangle$, so $m = n$.

Def 11: The completion of r on A is the partial order on the set $A \cup \{\lambda, \rho\} = A^*$ defined by

$r^* = C(p(r) \cup X \cup Y)$, where

$X = \{(\lambda, a) \mid P_r(a) = \emptyset\}$

$Y = \{(a, \rho) \mid F_r(a) = \emptyset\}$

Def 12: A <u>lattice</u> is a partial order such that the operations inf and sup on all pairs of elements are well defined.

Def 13: A <u>path</u> from a to b (both in A) is an n-tuple $(c_1, \ldots, c_n)$ such that

1) $c_1 = a; c_n = b$

2) $\forall i \leq n-1, (c_i, c_{i+1}) \in p(r)$

Def 14: $X \subset A$ is a <u>cluster</u> if

1) $o(X) \geq 2$

2) $x, y \in X \Rightarrow P_r(x) = P_r(y)$ and $F_r(x) = F_r(y)$

Def 15: $T \subset r$ is a <u>tangle</u> if

1) There is a least element α and a greatest element β in T

2) $\exists\, b_1, b_2, \ldots b_5 \in T'$ such that

a) $b_1 < b_2, b_3, b_4, b_5$

b) $b_2, b_3, < b_4$

c) $b_2 < b_5$

d) no relation in r holds between b_2, b_3 or b_4, b_5

e) if $\sup\{b_2, b_3\}$ and $\inf\{b_4, b_5\}$ exist, $\sup \not< \inf$

<u>Theorem 1</u>: If r is a partial order on A then r^* is a lattice or r^* contains a tangle.

<u>Proof:</u> Suppose r^* is not a lattice. Then since $a \in A \Rightarrow \lambda < a < \rho$ the only way r^* can fail to be a lattice if if $\exists$ a, b, $\in$ A such that $\{c \in A \mid c \geq a, c \geq b\}$ contains at least two minimal elements, or if a similar statement holds about $\{c \in A \mid c \leq a, c \leq b\}$

Assume without loss of generality that c_1 and c_2 are distinct minimal elements of $\{c \in A \mid c \geq a, c \geq b\}$. I now claim that there is a tangle in r^*.

Proof of Claim: Let $\alpha = \lambda$, $\beta = \rho$, $b_1 = a$, $b_2 = a$, $b_3 = b$, $b_4 = c_1$, $b_5 = c_2$.

Then we have a) $b_1 < b_2, b_3, b_4, b_5$ trivially

b) $b_2, b_3 < b_4$ since $b_4 \in \{c \in \Lambda \mid c \geq b, c, \in \Lambda\}$ and the intersection of this set with $\{a,b\}$ is $\emptyset$ because if a (say) is in the set then it would be the only minimal element and we have assumed there are at least two.

c) $b_2 < b_5$ for the same reason.

d) No relation holds between b_2 and b_3 because otherwise the greater one would be in $\{c \in \Lambda \mid c \geq b, c \geq a\}$. No relation holds between b_4 and b_5 since they are distinct minimal elements of a set.

e) $\sup \{b_2, b_3\}$ does not exist by assumption.

Thus if r* is not a lattice it contains a tangle.

Def 16: $X \subset Y$ is a segment iff:

1) Y is a lattice.

2) $\exists\ \alpha, \beta \in X'$ such that

a) $x \in X' \Rightarrow \alpha \leq x \leq \beta$

b) $\alpha \leq x \leq \beta \Rightarrow x \in X'$ or x is a member of a base chain of Y connecting α and β.

Def 17: A segment X is closed under r if $\forall\ x \in X'$,

$(y,x) \in p(r)$ or $(x,y) \in p(r) \Rightarrow x = \sup X'$ or $x = \inf X'$ or $y \in X'$.

Theorem 2: If X is a closed segment of Y and X contains a tangle, cluster, or ready pair of merge points then Y contains the same tangle, cluster or ready pair.

Proof: 1) tangle: Since $Y \subset X$, the X-tangle would fail to be a Y-tangle if b_2 and b_3 or b_4 and b_5 were comparable in Y. But for this to be true (say for example $b_2 <_Y b_3$) then $\exists\ y \in Y' - X'$ $b_2 <_Y y <_Y b_3$.

But since neither b_2 nor b_3 can be inf X' or sup X' this is impossible. The only other possibility is that inf $\{b_4, b_5\} >_X \inf_X \{b_4, b_5\}$ or $\sup_Y \{b_2, b_3\} <_X \sup_X \{b_2, b_3\}$ and this would violate closure.

2) cluster: an X-cluster C could fail to be a Y-cluster only if:

a) $\exists\ y \in Y'-X'$, $c_1, c_2 \in C$ such that $c_1 < y < c_2$, or

b) $\exists\ y \in Y'-X'$, $c_1, c_2 \in C \ni\ y < c_1$ and $y \not< c_2$ or $y > c_1$ and $y \not> c_2$.

But either of these conditions would contradict the closedness of X.

3) ready pair: If (X_1, X_2) is X-ready then the only way it can fail to be Y-ready is if one of the chains connecting X_1 and X_2 is not a base chain of Y. But this can only happen if $\exists\ y \in Y-X'$, $b_i \in < b_1, \ldots, b_n >$ connecting X_1 and X_2 such that $(y, b_i) \in p(Y)$ or $(b_i, y) \in p(Y)$. But this contradicts the closedness of X.

Theorem 3: If a lattice contains a non-trivial (i.e. one that does not consist of a single chain) segment which contains a tangle, cluster or ready pair then the lattice also contains a tangle, cluster or ready pair (not necessarily the same one). Let r be a lattice, X a segment of r.

Proof: Consider $Z = \{z \in X' \mid \inf_{r|X'} \mu(r|X) \leq z \leq \sup_{r|X'} \mu(r|X)$.

$r|Z$ is a segment of r and any tangle, cluster or ready pair in X must also be in $r|Z$. If $r|Z$ is closed under r then by Theorem 2 we are done. Therefore, assume $r|Z$ is not closed under r, and without loss of generality assume $\exists\ y \in r' - Z$, $z \in Z \ni (z,y) \in r$. Then let $b_1 = \inf Z$, $b_2 = z$, $b_3 =$ any element of Z not comparable or equal to Z, $b_4 = \sup Z$, $b_5 = y$. This is clearly an r-tangle. The only problem which could arise is if z above is comparable to every other point of Z. If not all elements of Z which connect to the outside (i.e. violate closure) have this property then we simply

choose one which does not have it and we are done. Thus assume $z_o \in Z$, $y \in r' - Z$ with $(y, z_o) \in p(r)$ or $(z_o, y) \in p(r) \Rightarrow \forall z \in Z - \{z_o\}$, $(z_o z) \in r|Z$ or $(z, z_o) \in r|Z$.

Let the elements of Z satisfying this condition be $Z^* = \{z_1, z_2, \ldots z_k\}$ and assume $z_1 < z_2 < \ldots < z_k$.

We know that if any (z_i, z_j) is ready in $r|Z$ then by Theorem 2 we are done, so at most one base chain of $r|Z$ connects any two elements of Z^*. Now let $Z^o = \{z \in Z \mid z \leq z_1\}$; $Z^i = \{z \in Z \mid z_i \leq z_{i+1}\}$, $1 \leq i \leq k-1$; $Z^k = \{z \in Z \mid z_k \leq z\}$.

It is clear that the $r|Z^i$ are segments of r, and I claim now that they are closed under r. For assume $\exists\ y \in r' - Z^i$, $z_o \in Z^i \ni (y, z_o) \in p(r)$ or $(z_o, y) \in p(r)$ but y is not comparable with either $\inf Z^i$ or $\sup Z^i$. Then, since z_o cannot be a point violating closure of Z, $y \in Z$. But $\forall\ z \in Z$, $\sup Z^i$ and $\inf Z^i$ are comparable to z. Hence the Z^i are closed under r.

I now claim further that any tangle, cluster or ready pair of Z must lie wholly within one of the Z^i.

a) <u>ready pair:</u> Assume $\{a, b\} \subset Z$ is ready and $a < b$, $a \in Z^i$. If $b \notin Z^i$ then a is an end-point of Z^i since the Z^i are closed. But now let $b \in Z^j$, so b must be an endpoint of Z^j. But by construction of the Z^i, no pair of endpoints can be ready unless they are endpoints of the same Z^i.

b) <u>tangle</u>: In this case we will modify the claim to assert that if there is a tangle in Z then one must lie wholly within a Z^i. To show this it is sufficient to show that in a Z-tangle b_2, b_3, b_4 and b_5 are in the same Z^i for none of them may be an endpoint, so the endpoints of the Z^i will do for α and β.

Thus assume there is a Z-tangle. Clearly b_2 and b_4 are in the same Z^i as b_3 and b_5 respectively, for any element of one Z^i is comparable

to any element of another. Also, since sup $\{b_2, b_3\} \not< \inf \{b_4, b_5\}$ b_5 is in the same Z^i as the sup. But this is the same as the one b_2 and b_3 are in so b_2, b_3, b_4 and b_5 are all in the same Z^i.

c) cluster: Since no two elements of a cluster are comparable they all must be in the same Z^i.

Now since the Z^i are closed under r, any tangle, cluster or ready pair in a Z^i is one in r. But any one in X is one in a Z^i. Hence anyone in X is one in r and we are done.

Procedure:

The purpose of this procedure is to construct an ascending sequence of partial orders on A, $(W_1, \ldots W_n)$, together with a sequence of numbers $(T_1, \ldots, T_n)$ with the property that if N_i is the number of linear orders on the set W_i, T_i N_i is a constant for all i.

To do this we will define set functions v_1 and v_2 defined on partial orders of A. The range of v_1 will be partial orders of A and the range of v_2 will be the natural numbers.

Now:

Given A, $r \subset A \times A$ an irreflexive partial order:

$W_1 = r$; $T_1 = 1$

$W_2 = r \cup <a_1, \ldots, a_n>$ where $<a_1, \ldots, a_n>$ is any linear order of the elements of $A - r'$; $T_2 = [o(A-r')]!$

$W_3 = W_2^*$; $T_3 = T_2$

for $n > 3$, $W_n = v_1 (W_{n-1})$; $T_n = [v_2(W_{n-1})] T_{n-1}$

Definition of v_1, v_2 (argument will be called r):

1) If there is an r-cluster $X \subset A$, then if $X = \{x_1, \ldots, x_n\}$

$$v_1 (r) = C[r | (A - X) \cup \{(a, x_1) | a \in P_r (X)\} \cup \{(x_n, b) | b \in F_r (X)\} \cup <x_1, \ldots, x_n>]$$

$$v_2 (r) = n!$$

2) If there is no cluster contained in r but $\exists\ p_1, p_2 \in \mu(r) \setminus (p_1, p_2)$ is ready, then if $x_1, x_2, \ldots x_n$ are all the base chains of r connecting p_1 and p_2,

$$v_1(r) = C\ [\ (p(r) - (\{(\sup X_i'\ ,\ p_2) \mid 1 \leqq i \leqq n-1\} \cup \{(p_1,\ \inf x_i') \mid 2 \leqq i \leqq n\})) \cup \{(\sup x_i',\ \inf x_{i+1}') \mid 1 \leqq i \leqq n-1\}\ ]$$

$$v_2(r) = \left| \begin{array}{c} \sum_{i=1}^{n} \ell(x_i)! \\ \prod_{i=1}^{n} [\ell(x_i)!] \end{array} \right|$$

3) If there is no cluster or ready pair, then

a) r consists of one chain, in which case $v_1(r) = r$, $v_2(r) = 1$, or

b) There is a tangle $T \subset r$ (see Theorem 4 below).

In case (b) consider the set Y of base chains of r which are contained in T and which meet the following conditions:

1) $y \in Y \Rightarrow d(T) > d(T|(T'-y'))$

2) $y \in Y$ = there is a path from X_y to Z_y containing no element of y'.

Note: $\{X_y\} = P_r(y), \{Z_y\} = F_r(y')$

Now denote the paths from X_y to Z_y by π^y, $1 \leq i \leq k_y$

For each y partition the set of paths into equivalence classes by number of merge points on each path. Choose from the class with the fewest merge points any path which has the maximal number of points of all paths in the class.

Call this path π_y.

Let $l(y)$ be the length of chain y

$m(y)$ be this number of merge points on π_y

$$g(y) = \frac{[l(y) + m(y)]!}{[l(y)]![m(y)]!}$$

Choose $y^* \in Y$ such that $g(y^*)$ is minimal.

Now considering only the merge points in π_{y^*} we have $g(y^*)$ order-preserving permutations of these with the points of y.

In other words there are $g(y^*)$ linear orders on the set $y^{*\prime} \cup [\mu(r) \cap \pi_{y^*}]$ which are consistent with r on this set.

Let $\mu(r) \cap \pi_{y^*}$ be $\{m_1, \ldots m_n\}$, $n = m(y^*)$

$y^* = \langle z_1, \ldots z_i \rangle$, $i = l(y^*)$

$\{\Lambda_1, \ldots, \Lambda_{g(y^*)}\}$ be the linear orders mentioned above.

Let $E_0^j = o(\{ z \in y^{*\prime} \mid (z, m_1) \in \Lambda_j \})$

$E_n^j = o(\{ z \in y^{*\prime} \mid (m_n, z) \in \Lambda_j \})$

$E_k^j = o(\{ z \in y^{*\prime} \mid (m_k, z), (z, m_{k+1}) \in \Lambda_j \}), 1 \leq k \leq n-1$

$F_0 = o(\{ f \in \pi'_{y^*} \mid (f, m_1) \in r \})$

$F_n = o(\{ f \in \pi_{y^*} \mid (m_n, f) \in r \})$

$F_k = o(\{ f \in \pi'_{y^*} \mid (m_k, f), (f, m_{k+1}) \in r \}), 1 \leq k \leq n-1$

The sets are partitions of y^* and π_{y^*} representing the points between successive pairs of merge points.

Now, if L is any linear order on T',

$$v_1(r) = (r-T) \cup L$$

$$\sum_{j=1}^{g(y^*)} \left[\left(\prod_{k=0}^{N} \begin{bmatrix} E_k^j + F_k \\ F_k \end{bmatrix} \right) v_2 ([T-y^*] \cup \Lambda) \right]$$

Theorem 4: If r is a partial order on A such that $r' = A$ and if r is not a linear order, then r^* must have at least one cluster, tangle or ready pair of merge points.

Proof: We proceed by induction or the number of base chains of r^*.

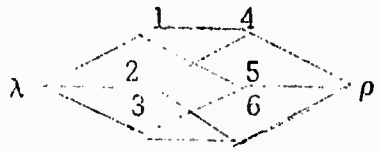
a) There is no base chain of r^*.

Thus, since $r' = A$, all members of A are merge points of r^*. We will show there is a cluster or tangle in r^*. We may assume r^* is a lattice or we would be done by Theorem 1. Consider $F_{r^*}(\lambda)$. Each element of this set has at least two successors because they are all merge points and they have only one predecessor (λ). We know $a, b \in F_{r^*}(\lambda) \Rightarrow F_{r^*}(b) \neq F_{r^*}(a)$ because r^* is a lattice. Now assume $\exists$ a, b $\in F_{r^*}(\lambda)$ $F_{r^*}(a) \cap F_{r^*}(b) \neq \emptyset$. Since r^* is a lattice this means $\exists$ c $\ni F_{r^*}(a) \cap F_{r^*}(b) = \{c\}$. Let $b_1 = \lambda$, $b_2 = a$, $b_3 = b$, $b_4 = c$, $b_5 \in F_{r^*}(a) - \{c\}$.

It is straightforward to verify that this defines a tangle. All we needed about a and b, however, was that $\Gamma_{r^*}(a) \cap \Gamma_{r^*}(b) \neq \emptyset$ and $P_{r^*}(a) \cap P_{r^*}(b) \neq \emptyset$. We will now show that there are always an a and b meeting these conditions.

Let M = maximum length of all paths from λ to ρ : we will proceed by induction on M.

If M ≤ 2 then we cannot have a lattice with all merge points, so let M = 3.



The above graph is a lattice of all merge points with M = 3.

Now consider $\Gamma_{r^*}(\lambda)$. This has at least two elements because otherwise $\exists$ ac $P_{r^*}(\rho)$ with only one preceder and one follower and hence violating our conditions. Now consider $\Gamma_{r^*}(a)$, ac $F_{r^*}(\lambda)$. $\rho \notin F_{r^*}(a)$ because a would then not be a merge point. Also, $o(F_{r^*}(a)) > 1$ for the same reason. Now $F_{r^*}(F_{r^*}(a)) = \{\rho\}$, so $F_{r^*}(a)$ is a cluster unless $\exists$ b ∈ $F_{r^*}(\lambda)$, c ∈ $F_{r^*}(a)$, b ≠ a, ∃ c ∈ $\Gamma_{r^*}(b)$. But this would mean that $F_{r^*}(b) \cap F_{r^*}(a) \neq \emptyset$ and we would have a tangle.

Therefore, assume our hypothesis is true for 3 ≤ M ≤ k. We wish to prove it for M = k + 1.

Let us therefore suppose that we have r* ∋ M = k + 1, $\mu(r^*) \supset A$. Consider $F_{r^*}(\lambda)$. We may assume that $o(F_{r^*}(\lambda)) > 1$ for otherwise r* | [A* - $F_{r^*}(\lambda)$] would have a tangle, cluster, or ready pair by our induction hypothesis, hence so would r* | A* - {λ}, and hence so would r*. We may further assume that a, bc $F_{r^*}(\lambda) \Rightarrow F_{r^*}(\lambda) \cap F_{r^*}(b) = \emptyset$, for if not we know we would have a tangle and we would be done.

Let B = A* - {a ∈ A | $\exists$ bc $F_{r^*}(\lambda) \supset$ ac $F_{r^*}(b)$}. If we now consider r* | B we have a partial order such that M = k. Thus if r* | B contains no non-merge points it will contain a cluster or tangle. But the only way for this to not happen is if a, bc A*-B ∋ $F_{r^*}(a) \cap F_{r^*}(b) = \emptyset$. But then we would have a

unless this set is empty. But if this set is empty then a cannot be a merge point. Hence we may assume $r^*|B$ has a tangle or a cluster.

If $r^*|B$ contains a tangle then this is also an r^* tangle unless the "not comparable" condition on b_2, b_3 and b_4, b_5 fail or if a new sup for b_2 and b_3 is less than a new inf for b_4 and b_5. But the first case cannot happen by the definition of a restriction of a partial order. For the second case to happen we would need $\sup_{r^*} \{b_2, b_3\} = \inf_{r^*} \{b_4, b_5\} \in A^* - B$.

But this cannot happen since we are assuming that $F_{r^*}(b_2)$ and $F_{r^*}(b_3)$ are disjoint if b_2, b_3, $\in F_{r^*}(\lambda)$ which would be a necessary condition for sup $\{b_2, b_3\} \in A^* - B$.

Hence if $r^*|B$ contains a tangle so does r^*.

Now $r^*|B$ cannot contain a cluster unless it contains a tangle because it is a lattice if it has no tangle, and every point is a merge point. In a lattice every cluster has a unique follower and a unique predecessor and hence its elements cannot be merge points.

Hence if r^* contains only merge points it contains a cluster or a tangle.

Now assume that if K is the number of base chains of a lattice. $K \leq p$ means that the lattice has a cluster, tangle, or ready pair. Let r* have p+1 base chains. Consider any right r*-merge point m and the segment $M = r^* |\{y | y \geq m\}$. If M has p or fewer base chains then it has a tangle, cluster, or ready pair, and by Theorem 3 so does r*. If M has at least p+1 base chains consider the set of right merge points of M greater than m. Assuming this set is non-empty let m_1 be a minimal element of it and let $M_1 = r^* |\{y | y \geq m_1\}$. $M_1 \subset M$ since $m_1 > m$ so $y > m_1 \Rightarrow y > m$. If M_1 has p or fewer base chains then apply Theorem 3 to show r* has a cluster, tangle, or ready pair, so assume M_1 has at least p+1 base chains. Proceeding as above we obtain a sequence of non-trivial segments of r*, each one properly contained in the preceding one. Since r* is finite this sequence must terminate, but it can do so only if

1) for some i, M_i has p or fewer base chains, or
2) for some i, M_i contains no right merge points other than m_i.

In case I an application of Theorem 3 shows that r* has a cluster, tangle, or ready pair, so let us consider case II. Let L be the set of left merge points of M_i, and let x be a minimal element of L. $x \neq m_i$ since m_i is not a left merge point of M_i, so $x > m_i$. Also, $\exists$ a, b $\in M_i \ni m_i < a < x$ and $m_i < b < x$ and a, b are not comparable in M_i. This is true because $(m_i, x) \notin M_i$ and by the minimality of x: Thus there is a path from m_i to x containing a and no merge points, and a path containing b and no merge points (by the minimality of x). Since a and b are not comparable they cannot be on the same base chain, so by Lemma 4 there are two base chains connecting m_i and x, so (m_i, x) is ready, and by Theorem 3 r* contains a cluster, tangle, or ready pair. Our induction is now complete. Hence, <u>the completion of any non-trivial partial order contains a cluster, tangle, or ready pair of merge points.</u>

Q.E.D.

APPENDIX C: Determining the Boundries of Optimization Effectiveness

Our approach is to assume that a balanced program exists, move the CU interchangeable instructions to a single PE, and compare execution time. We first consider a block of code in which all the instructions are interchangeable. For present purposes we further assume that each interchangeable instruction takes approximately the same amount of time to execute in its respective processor. Call average instruction times Tcu and Tpe. Let Npe = the number of PE instructions and Ncu = the number of CU instructions. Assume a balanced segment of code executed in time T. Since execution is balanced: $T = (Npe)(Tpe) = (Ncu)(Tcu) + Npe$; that is, the product of the number of instructions and average instruction time in the PE's is equal to that same product in the CU plus an additional tick to decode each PE instruction

Hence; $$Ncu = (Npe)\frac{(Tpe-1)}{Tcu}$$

If all the instructions were executed in the PE's, then the total execution time would be:

$$T = \left[Npe + (Npe)\frac{(Tpe-1)}{Tcu}\right](Tpe)$$

The ratio of balanced to single processor execution is:

$$\frac{(Npe)(Tpe)}{(Npe)(Tpe)\left[1 + \frac{(Tpe-1)}{Tcu}\right]} = \frac{Tcu}{Tcu+Tpe-1}$$

The savings factor is: $$1 - \frac{Tcu}{Tcu+Tpe-1} = \frac{Tpe-1}{Tcu+Tpe-1}$$

An examination of the second case, namely that a segment contains PE exclusive instructions (i.e. SIM assignments) and a sufficient number of inter-changeable instructions to balance them, yields the same result. This is due to the fact that regardless of what is being executed in the PE's, the interchangeable instructions removed from the CU and placed in a single PE will increase execution by the same margin. Hence allocation and relocation as equally effective techniques.

Estimates of execution savings are based on a number of simplifying assumptions.

With regard to average execution time, some instructions are combined. For example, a PE 'load' is interpreted as a load to the R register and a route. CU average execution is determined for two cases; optimized and unoptimized. For optimized execution it is assumed that a single

load from PE memory is required for eight references to local memory. In the unoptimized case, it is assumed that each reference to local memory requires an additional load from PE memory. In either case, a 'load' is the combined instructions.

Interchangeable Operation	PE	CU Optimized	CU Unoptimized
LOAD	3	4	11
OPERATION {ADD, SUB, OR, Etc.}	3	4	11
STORE	3	4	11
Estimated Savings	-	33%	15%

[6]

REFERENCES

1. Allen, F. E. Program optimization. In Halpern, Mark I., and Shaw, Christopher J. (eds.), Annual Review in Automatic Programming 5, 239-307. Pergamon Press, Oxford, 1969.

2. Ramamoorthy, C. V., and Gonzalez, M. J. A survey of techniques for recognizing parallel processable streams in computer programs. Proc. AFIPS 1969 Fall Joint Comput. Conf. 35, 1-15.

3. Warshall, Stephen. A theorem on Boolean matrices. J. ACM 9, 1 (Jan. 1962), 11-12.

4. Lowe, Thomas C. Analysis of Boolean program models for time-shared, paged environments. Comm. ACM 12, 4 (Apr. 1969), 199-205.

5. Shapiro, Robert M., and Saint, Harry. The Representation of Algorithms. RADC-TR-69-313, Volume II, Final Technical Report. Applied Data Research, Inc., Wakefield, Mass., Sept. 1969.

6. Burroughs Corporation. ILLIAC IV Systems Characteristics and Programming Manual. IL4-PM1. Burroughs Corporation, Paoli, Penna., June 1969.